Ingeniería del software

Ingeniería del software

Séptima edición

IAN SOMMERVILLE

Traducción

María Isabel Alfonso Galipienso
Antonio Botía Martínez
Francisco Mora Lizán
José Pascual Trigueros Jover
Departamento Ciencia de la Computación e Inteligencia Artificial
Universidad de Alicante

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • São Paulo • Reading, Massachusetts • Harlow, England

Datos de catalogación bibliográfica

INGENIERÍA DEL SOFTWARE. Séptima edición
Ian Sommerville

PEARSON EDUCACIÓN, S.A., Madrid, 2005

ISBN: 84-7829-074-5

MATERIA: Informática 681.3

Formato: 195 × 250 mm Páginas: 712

INGENIERÍA DEL SOFTWARE. Séptima edición
Ian Sommerville

ISBN: 84-7829-074-5
Depósito Legal: M-31.467-2005

PEARSON ADDISON WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Equipo editorial:
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

Equipo de producción:
Director: José Antonio Clares
Técnico: José Antonio Hernán

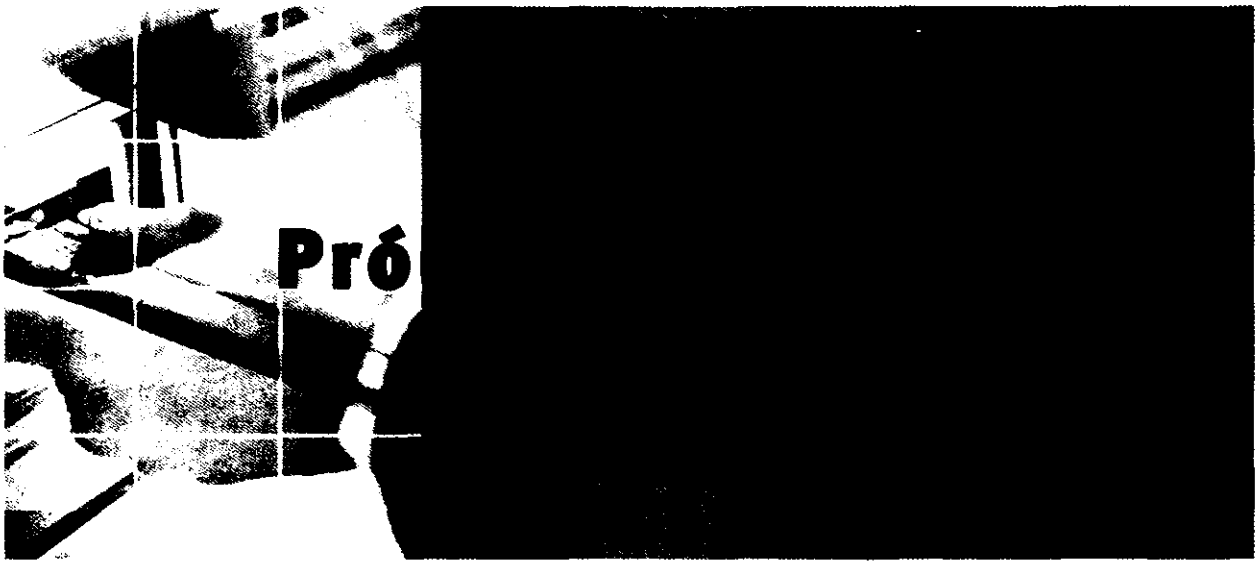
Diseño de cubierta: Equipo de diseño de Pearson Educación, S.A.

Composición: COPIBOOK, S.L.

Impreso por: TOP PRINTER PLUS, S. L. L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos



La primera edición de este libro de ingeniería del software fue publicada hace más de veinte años. Aquella edición fue escrita utilizando un terminal de texto conectado a una minicomputadora (un PDP-11) que posiblemente costaba cerca de 50.000 $. Yo he escrito esta edición desde un portátil con conexión inalámbrica que cuesta menos de 2.000 $ y mucho más potente que aquel PDP-11. El software más común era el software para mainframes, pero las computadoras personales estaban a punto de aparecer. Ninguno de nosotros imaginó el nivel de difusión que éstas iban a tener ni el cambio que este hecho iba a producir en el mundo.

Los cambios en el hardware en los últimos veinte años han sido notables, y podría parecer que los cambios en el software han sido igual de significativos. Ciertamente, nuestra capacidad para construir sistemas grandes y complejos ha mejorado drásticamente. Nuestros servicios e infraestructuras nacionales —energía, comunicaciones y transporte— dependen de sistemas informáticos muy grandes, complejos y fiables. En la construcción de sistemas software se mezclan muchas tecnologías —J2EE, .NET, EJB, SAP, BPEL4WS, SOAP, CBSE— que permiten que aplicaciones grandes basadas en Web sean desarrolladas mucho más rápido que en el pasado.

Sin embargo, a pesar de los cambios que ha habido en las dos últimas décadas, cuando nosotros miramos más allá de las tecnologías, hacia los procesos fundamentales de la ingeniería del software, éstos se han mantenido igual. Nosotros reconocimos hace veinte años que el modelo en cascada tenía problemas serios, pero un examen publicado en diciembre de 2003 por *IEEE* mostraba que más de un 40% de las compañías siguen utilizando esta aproximación. El testeo sigue siendo la técnica de validación dominante, a pesar de que otras técnicas, como las inspecciones, han sido utilizadas de una forma más efectiva desde mediados de los años 70. Las herramientas CASE, a pesar de estar basadas ahora en UML, siguen siendo básicamente editores gráficos con alguna funcionalidad para chequear y generar código.

Nuestros actuales métodos y técnicas de ingeniería del software han hecho que la construcción de sistemas grandes y complejos sea mejor. A pesar de ello, sigue siendo habitual encontrar proyectos que se retrasan, que sobrepasan el presupuesto o que se entregan sin satisfacer las necesidades de los clientes. Mientras estaba escribiendo este libro, se divulgó una investigación del gobierno en Reino Unido, sobre un proyecto para proveer a los juzgados de un sistema software para casos de delincuencia menor. El coste del sistema fue estimado en 156 millones de libras y fue planificado para ser entregado en el año 2001. En 2004, el coste había subido a 390 millones de libras y no estaba totalmente operativo. Hay, por lo tanto, una necesidad imperiosa de educación en ingeniería del software.

En los últimos años, el desarrollo más significativo en ingeniería del software ha sido la aparición de UML como estándar para la descripción de sistemas orientados a objetos, y el desarrollo de métodos ágiles como la programación extrema. Los métodos ágiles están permitiendo el desarrollo rápido de sistemas, explícitamente implican al usuario en el equipo de trabajo y reducen el papeleo y la burocracia en el proceso software. A pesar de lo que algunos críticos sostienen, pienso que estas aproximaciones encarnan buenas prácticas de ingeniería del software. Ellas tienen unos procesos bien definidos, prestan atención a la especificación del sistema y a los requerimientos del usuario, y tienen estándares de alta calidad.

No obstante, esta revisión no pretende ser un texto sobre métodos ágiles. Prefiero centrarme en los procesos básicos de ingeniería del software —especificación, diseño, implementación, verificación, y validación y gestión—. Es necesario entender estos procesos y las técnicas asociadas para decidir si los métodos ágiles son la estrategia de desarrollo más adecuada y cómo adaptar los métodos a una situación particular. Un tema dominante en el libro son los sistemas críticos —sistemas en los que los fallos de funcionamiento tienen consecuencias nefastas y donde la seguridad del sistema es crítica. En cada parte del libro, estudiaremos las técnicas específicas de ingeniería del software que son relevantes para la construcción sistemas críticos.

Inevitablemente, los libros reflejan las opiniones y prejuicios de sus autores. Algunos lectores estarán en desacuerdo con mis opiniones y con mi elección del material. Este desacuerdo es un reflejo de la diversidad de disciplinas y es esencial para su evolución. No obstante, yo espero que a todos los ingenieros de software y estudiantes de ingeniería del software les resulte interesante.

Estructura del libro

La estructura del libro está basada en los procesos fundamentales de la ingeniería del software. Está organizado en seis partes, con varios capítulos en cada parte:

Parte 1: Introduce la ingeniería del software, situándola en un amplio contexto de sistemas y presentando las nociones de procesos y gestión de ingeniería del software.
Parte 2: Trata los procesos, técnicas y documentación asociados con los requerimientos de ingeniería. Incluye un estudio sobre los requerimientos software, modelado de sistemas, especificación formal y técnicas para especificar la fiabilidad.
Parte 3: Esta parte está dedicada al diseño de software y a los procesos de diseño. Tres de los seis capítulos se centran en el importante tema de las arquitecturas software. Otros temas incluyen diseño orientado a objetos, diseño de sistemas en tiempo real y diseño de interfaces de usuario.

Parte 4: Describe una serie de aproximaciones a la implementación, incluyendo métodos ágiles, reutilización, CBSE y desarrollo de sistemas críticos. Como los cambios son una parte importante de la implementación, he integrado temas de evolución y mantenimiento en esta parte.
Parte 5: Se centra en temas de verificación y validación. Incluye capítulos de validación y verificación estática, testeo y validación de sistemas críticos.
Parte 6: La parte final abarca una serie de temas de gestión: gestión de personal, estimación de costes, gestión de calidad, procesos de mejora y gestión de cambios.

En la introducción de cada parte, expondré la estructura y organización con mayor detalle.

Cambios en la 6.ª edición

Hay cambios importantes, relativos a la organización y contenido, respecto a la edición previa. He incluido cuatro capítulos nuevos y he hecho una importante revisión en otros once capítulos. Todos los otros capítulos han sido actualizados, incorporando convenientemente nuevo material.

Más y más sistemas tienen altos requerimientos de disponibilidad y fiabilidad, y espero que nosotros tomemos la fiabilidad como un conductor básico en la ingeniería del software. Por esta razón, los capítulos de sistemas críticos han sido integrados en otras secciones. Para evitar que el volumen del libro sea excesivo, he reducido la cantidad de material sobre mantenimiento del software y he integrado los temas de mantenimiento y evolución del software en otros capítulos. Hay dos casos de estudio —uno sobre la gestión de documentos de una biblioteca y otro de un sistema médico—, los cuales presento en diferentes capítulos.

El material referente a los casos de estudio está señalado con iconos en los márgenes. La Tabla 1 resume los cambios, indicando con el número entre paréntesis el capítulo correspondiente en la 6.ª edición. En el sitio Web del libro se puede encontrar más información sobre estos cambios.

TABLA 1. Revisión de los capítulos

Capítulo 13:	Arquitectura de las aplicaciones.
Capítulo 17:	Desarrollo rápido de aplicaciones.
Capítulo 19:	Ingeniería del software basada en componentes.
Capítulo 21:	Evolución del software
Capítulo 2:	Sistemas socio-técnicos (2)
Capítulo 4:	Procesos software (3)
Capítulo 7:	Procesos de ingeniería de requerimientos (6)
Capítulo 9:	Especificación de sistemas críticos (17)
Capítulo 12:	Arquitecturas de sistemas distribuidos (11)
Capítulo 16:	Diseño de interface de usuario (15)
Capítulo 18:	Reutilización de código (14)
Capítulo 23:	Pruebas del software (20)
Capítulo 25:	Gestión de personal (22)
Capítulo 24:	Validación de sistemas críticos (21)
Capítulo 28:	Mejora de procesos (25)

Capítulo 1: Introducción (1)
Capítulo 3: Sistemas críticos (16)
Capítulo 5: Gestión de proyectos (4)
Capítulo 6: Requerimientos software (5)
Capítulo 8: Modelos de sistemas (7)
Capítulo 10: Especificación formal (9)
Capítulo 11: Diseño arquitectónico (10)
Capítulo 14: Diseño orientado a objetos (12)
Capítulo 15: Diseño de sistemas en tiempo real (13)
Capítulo 20: Implementación de sistemas críticos (18)
Capítulo 22: Verificación y validación (19)
Capítulo 26: Estimación de costes del software (23)
Capítulo 27: Gestión de calidad (24)
Capítulo 29: Gestión de configuraciones (29)

Construcción de prototipos software (8)
Sistemas heredados (26)
Cambios en el software (27)
Reingeniería del software (28)

Guía para el lector

Este libro está enfocado a estudiantes, graduados e ingenieros de la industria del software. Puede ser utilizado en cursos generales de ingeniería del software o en cursos específicos, como programación avanzada, especificación, diseño y gestión software. A los ingenieros de software que trabajan en la industria, este libro puede resultarles útil como lectura general y como actualización de sus conocimientos en temas particulares como ingeniería de requerimientos, diseño de la arquitectura, desarrollo de sistemas formales y mejora de procesos. Los ejemplos en el texto han sido utilizados como una vía práctica para reflejar el tipo de aplicaciones que los ingenieros deben desarrollar.

Usando el libro para enseñar

He diseñado el libro para que pueda ser utilizado en tres tipos de cursos de ingeniería.

1. *Cursos de introducción a la ingeniería del software* para estudiantes que no tienen experiencia en ingeniería del software. Se puede empezar con la sección introductoria y luego seleccionar capítulos de otras secciones del libro. Esto dará a los estudiantes una visión general de la materia con la oportunidad de hacer un estudio más detallado por parte de los alumnos interesados. Si el curso está basado en proyectos, los primeros capítulos proporcionarán suficiente material para comenzar el proyecto, y capítulos posteriores servirán para referenciar y dar más información del avance de su trabajo.

2. *Cursos de introducción o nivel medio sobre temas específicos de ingeniería del software*. El libro es válido para cursos de especificación de requerimientos, diseño de software, gestión de proyectos software, desarrollo de sistemas fiables y evolución del software. Cada parte puede servir tanto para cursos de introducción como intermedios en los diferentes temas. Así como cada capítulo tiene lecturas asociadas, he incluido en el sitio web información sobre otros artículos y libros relevantes.
3. *Cursos avanzados en ingeniería del software*. Los capítulos pueden servir como base para cursos específicos, pero deben ampliarse con lecturas complementarias que traten con mayor detalle los temas. Por ejemplo, yo imparto un módulo en Master en ingeniería de sistemas el cual se apoya en este material. He incluido detalles de este curso y de un curso en ingeniería de sistemas críticos en el sitio web.

La utilidad de un libro general como éste está en que puede utilizarse en diferentes cursos. En Lancaster, nosotros empleamos el texto en un curso de introducción a la ingeniería del software y en cursos de especificación, diseño y sistemas críticos. Cursos de ingeniería del software basada en componentes e ingeniería de sistemas utilizan el libro a lo largo de su impartición, junto con artículos complementarios que se distribuyen entre los alumnos. Tener un solo libro de texto da a los alumnos una visión coherente de la materia, y éstos no tienen que comprar varios libros.

Para reforzar la experiencia de aprendizaje de los estudiantes, he incluido un glosario de términos, con definiciones adicionales en el sitio web. Además, cada capítulo tiene:

- Unos objetivos claros presentados en la primera página.
- Una lista de los puntos clave tratados en el capítulo.
- Lecturas adicionales recomendadas —otros libros que están actualmente en impresión o artículos fácilmente accesibles (en mi sitio web puede encontrar un listado de otras lecturas y enlaces recomendados).
- Ejercicios, que incluyen ejercicios de diseño.

El proyecto denominado «Software Engineering Body of Knowledge» (http://swebok.org) fue establecido para definir las áreas clave de conocimiento técnico relevantes para los profesionales del software. Están organizadas bajo 10 epígrafes: requerimientos, diseño, construcción, prueba, mantenimiento, gestión de configuraciones, gestión, procesos, herramientas y métodos, y calidad. Mientras sería imposible incluir en un solo libro de texto todas las áreas de conocimiento propuestas por el proyecto SWEBOK, en este libro se tratan todas las áreas de alto nivel.

Páginas Web

El sitio web asociado a este libro es:

http://www.software-engin.com

Este sitio ofrece una amplia gama de material complementario de ingeniería del software. Desde aquí, usted puede acceder a las páginas web de soporte de este libro y de ediciones anteriores.

Ésta ha sido mi política, en versiones anteriores y en ésta, para mantener el número de enlaces web en el libro en un mínimo absoluto. La razón es que los enlaces web sufren muchos cambios y, una vez impreso el libro, son imposibles de actualizar. En consecuencia, la pági-

na web del libro incluye un gran número de enlaces a recursos y material relacionado con la ingeniería del software. Si usted los utiliza y encuentra problemas, por favor hágamelo saber y actualizaré esos enlaces.

Para dar soporte en el uso de este libro en cursos de ingeniería del software, he incluido una amplia variedad de material en el sitio web. En los enlaces al material para el docente, podrá encontrar:

- Presentaciones (PowerPoint y PDF) para todos los capítulos del libro.
- Cuestiones para cada capítulo.
- Casos de estudio.
- Sugerencias sobre proyectos.
- Descripciones sobre estructuras de cursos.
- Sugerencias sobre lecturas complementarias y enlaces a los recursos web de cada capítulo.
- Soluciones para los ejercicios asociados a cada capítulo y para las cuestiones (sólo profesor).

Sus sugerencias y comentarios sobre el libro y el sitio web serán bienvenidos. Puede contactar conmigo a través de ian@software-engin.com. Recomiendo que incluya [SE7] en el asunto del mensaje para evitar que los filtros antispam rechacen su mensaje. Siento no tener tiempo para ayudar a los estudiantes en su trabajo; por lo tanto, no me pregunten cómo resolver ningún problema del libro.

Reconocimientos

A lo largo de los años muchas personas han contribuido al desarrollo de este libro, y me gustaría dar las gracias a todos los que han comentado las ediciones anteriores y han hecho sugerencias constructivas (revisores, estudiantes, lectores que me han escrito). Al personal de la editorial y producción de Pearson Educación en Inglaterra y en Estados Unidos por su apoyo y ayuda, y producir el libro en un tiempo récord. Por lo tanto, gracias a Keith Mansfield, Patty Mahtani, Daniel Rausch, Carol Noble y Sharon Burkhardt por su ayuda y apoyo.

Finalmente, me gustaría dar las gracias a mi familia, que ha tolerado mi ausencia cuando el libro estaba comenzándose a escribir y mi frustración cuando las palabras no surgían. Y en especial a mi esposa, Anne, y a mis hijas, Ali y Jane, por su ayuda y apoyo.

Ian Sommerville,
febrero 2004

PARTE

I VISIÓN GENERAL

La estructura básica de este libro sigue los procesos esenciales del software de especificación, diseño, desarrollo, verificación y validación, y gestión. Sin embargo, más que caer inmediatamente en estos temas, he incluido esta sección de visión general para que pueda tener una idea amplia de la disciplina. Esta parte comprende los cinco primeros capítulos:

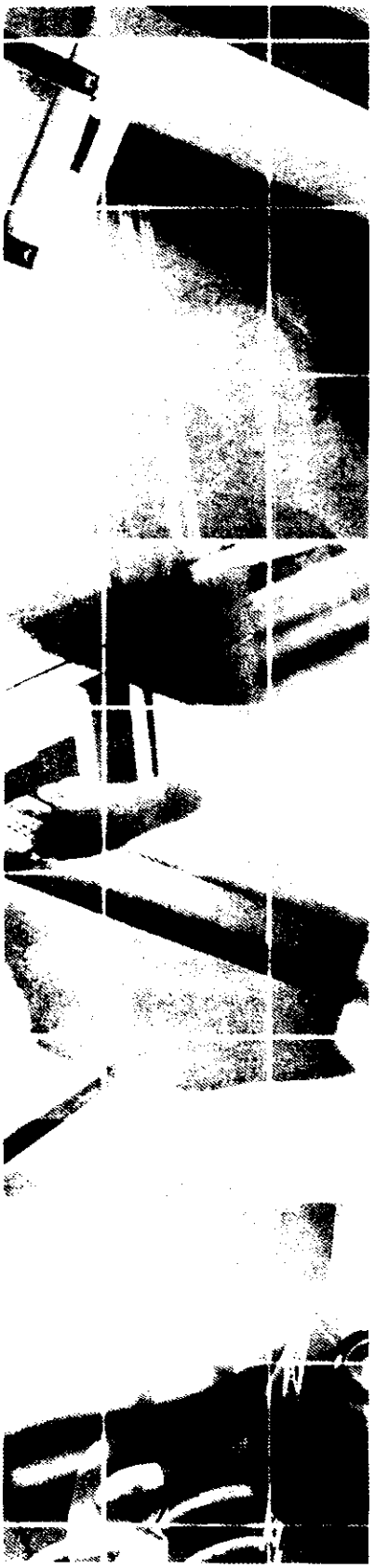
El Capítulo 1 es una introducción general a la ingeniería del software. Para hacerlo accesible y fácil de entender, lo he organizado usando una estructura de pregunta/respuesta donde planteo y respondo preguntas tales como «¿Qué es la ingeniería del software?». También introduzco el profesionalismo y la ética en este capítulo.

El Capítulo 2 presenta los sistemas socio-técnicos, un tema que creo es absolutamente esencial para los ingenieros de software. El software nunca es usado por sí solo, pero siempre es parte de un sistema mayor que incluye el hardware, el elemento humano y, a menudo, las organizaciones. Estos componentes influyen profundamente en los requerimientos y funcionamiento del software. En este capítulo se estudian las propiedades emergentes de los sistemas, los procesos de la ingeniería de sistemas y algunas de las formas en las que los asuntos organizacionales y humanos afectan a los sistemas de software.

El Capítulo 3 trata los «sistemas críticos». Los sistemas críticos son sistemas en los que un fallo de funcionamiento del software tiene graves consecuencias técnicas, económicas o humanas, y en los que la seguridad del sistema, la protección y la disponibilidad son requerimientos clave. Se incluyen capítulos sobre aspectos de sistemas críticos en cada parte del libro. En este capítulo, además, presento el primero de los casos de estudio del libro: el software para una bomba de insulina usado en el tratamiento de pacientes diabéticos.

Los tres primeros capítulos establecen el escenario de la ingeniería del software y el Capítulo 4 continúa con este propósito introduciendo el proceso del software y los modelos de procesos del software. Introduzco procesos básicos de la ingeniería del software, la materia del libro, en este capítulo. También trato brevemente el Proceso Unificado de Rational, el cual está enfocado al desarrollo de sistemas orientados a objetos. La última sección del capítulo explica cómo los procesos del software pueden ser apoyados con herramientas software automatizadas.

El Capítulo 5 aborda la gestión de proyectos. La gestión de proyectos es parte de todos los desarrollos profesionales de proyectos, y aquí describo la planificación del proyecto, la confección de agendas y la estimación de riesgos. Los estudiantes de un curso de ingeniería del software implicados en un proyecto estudiantil deberían encontrar aquí la información que necesitan para trazar gráficos de barras para un programa del proyecto y para la asignación de recursos.



1 Introducción

Objetivos

Los objetivos de este capítulo son introducir la ingeniería del software y proporcionar un marco para entender el resto del libro. Cuando haya leído este capítulo:

- comprenderá qué es la ingeniería del software y por qué es importante;
- conocerá las respuestas a las preguntas clave que proporcionan una introducción a la ingeniería del software;
- comprenderá algunos aspectos profesionales y de ética que son importantes para los ingenieros de software.

Contenidos

Actualmente casi todos los países dependen de complejos sistemas informáticos. Infraestructuras nacionales y utilidades dependen de sistemas informáticos, y la mayor parte de los productos eléctricos incluyen una computadora y software de control. La fabricación industrial y distribución está completamente informatizada, como el sistema financiero. Por lo tanto, producir software costeable es esencial para el funcionamiento de la economía nacional e internacional.

La ingeniería del software es una disciplina de la ingeniería cuya meta es el desarrollo costeable de sistemas de software. Éste es abstracto e intangible. No está restringido por materiales, o gobernado por leyes físicas o por procesos de manufactura. De alguna forma, esto simplifica la ingeniería del software ya que no existen limitaciones físicas del potencial del software. Sin embargo, esta falta de restricciones naturales significa que el software puede llegar a ser extremadamente complejo y, por lo tanto, muy difícil de entender.

La noción de *ingeniería del software* fue propuesta inicialmente en 1968 en una conferencia para discutir lo que en ese entonces se llamó la «crisis del software». Esta crisis del software fue el resultado de la introducción de las nuevas computadoras hardware basadas en circuitos integrados. Su poder hizo que las aplicaciones hasta ese entonces irrealizables fueran una propuesta factible. El software resultante fue de órdenes de magnitud más grande y más complejo que los sistemas de software previos.

La experiencia previa en la construcción de estos sistemas mostró que un enfoque informal para el desarrollo del software no era muy bueno. Los grandes proyectos a menudo tenían años de retraso. Costaban mucho más de lo presupuestado, eran irrealizables, difíciles de mantener y con un desempeño pobre. El desarrollo de software estaba en crisis. Los costos del hardware se tambaleaban mientras que los del software se incrementaban con rapidez. Se necesitaban nuevas técnicas y métodos para controlar la complejidad inherente a los sistemas grandes.

Estas técnicas han llegado a ser parte de la ingeniería del software y son ampliamente utilizadas. Sin embargo, cuanto más crezca nuestra capacidad para producir software, también lo hará la complejidad de los sistemas de software solicitados. Las nuevas tecnologías resultantes de la convergencia de las computadoras y de los sistemas de comunicación y complejas interfaces gráficas de usuario impusieron nuevas demandas a los ingenieros de software. Debido a que muchas compañías no aplican de forma efectiva las técnicas de la ingeniería del software, demasiados proyectos todavía producen software que es irrealizable, entregado tarde y sobrepresupuestado.

Se puede afirmar que hemos hecho enormes progresos desde 1968 y que el desarrollo de esta ingeniería ha mejorado considerablemente nuestro software. Comprendemos mucho mejor de las actividades involucradas en el desarrollo de software. Hemos desarrollado métodos efectivos de especificación, diseño e implementación del software. Las nuevas notaciones y herramientas reducen el esfuerzo requerido para producir sistemas grandes y complejos.

Ahora sabemos que no hay una enfoque «ideal» a la ingeniería del software. La amplia diversidad de diferentes tipos de sistemas y organizaciones que usan estos sistemas significa que necesitamos una diversidad de enfoques al desarrollo de software. Sin embargo, las nociones fundamentales de procesos y la organización del sistema son la base de todas estas técnicas, y éstas son la esencia de la ingeniería del software.

Los ingenieros de software pueden estar orgullosos de sus logros. Sin software complejo no habríamos explorado el espacio, no tendríamos Internet y telecomunicaciones modernas, y todas las formas de viajar serían más peligrosas y caras. Dicha ingeniería ha hecho enormes contribuciones, y no cabe dudad de que, en cuanto la disciplina madure, su contribución en el siglo XXI será aún más grande.

1.1 Preguntas frecuentes sobre la ingeniería del software

Esta sección se ha diseñado para resolver algunas preguntas fundamentales sobre la ingeniería del software y para proporcionar algunos de mis puntos de vista sobre la disciplina. El formato que he utilizado es el de «lista de preguntas frecuentes». Este enfoque se emplea comúnmente en los grupos de noticias de Internet para proveer a los recién llegados de las respuestas a las preguntas frecuentes. Creo que es una manera muy efectiva de dar una introducción sucinta al tema de la ingeniería del software.

Las preguntas que se contestan en esta sección se muestran en la Figura 1.1.

1.1.1 ¿Qué es software?

Muchas personas asocian el término *software* con los programas de computadora. Sin embargo, yo prefiero una definición más amplia donde el software no son sólo programas, sino todos los documentos asociados y la configuración de datos que se necesitan para hacer que estos programas operen de manera correcta. Por lo general, un sistema de software consiste en diversos programas independientes, archivos de configuración que se utilizan para ejecu-

¿Qué es software?	Programas de ordenador y la documentación asociada. Los productos de software se pueden desarrollar para algún cliente en particular o para un mercado general.
¿Qué es la ingeniería del software?	La ingeniería del software es una disciplina de ingeniería que comprende todos los aspectos de la producción de software. ¿Cuál es la diferencia entre ingeniería del software y ciencia de la computación? La ciencia de la computación comprende la teoría y los fundamentos; la ingeniería del software comprende las formas prácticas para desarrollar y entregar un software útil. ¿Cuál es la diferencia entre ingeniería del software e ingeniería de sistemas? La ingeniería de sistemas se refiere a todos los aspectos del desarrollo de sistemas informáticos, incluyendo hardware, software e ingeniería de procesos. La ingeniería del software es parte de este proceso.
¿Qué es un proceso del software?	Un conjunto de actividades cuya meta es el desarrollo o evolución del software.
¿Qué es un modelo de procesos del software?	Una representación simplificada de un proceso del software, presentada desde una perspectiva específica.
¿Cuáles son los costos de la ingeniería del software?	A grandes rasgos, el 60 % de los costos son de desarrollo, el 40 % restante son de pruebas. En el caso del software personalizado, los costos de evolución a menudo exceden los de desarrollo.
¿Qué son los métodos de la ingeniería del software?	Enfoques estructurados para el desarrollo de software que incluyen modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos.
¿Qué es CASE (Ingeniería del Software Asistida por Ordenador)?	Sistemas de software que intentan proporcionar ayuda automatizada a las actividades del proceso del software. Los sistemas CASE a menudo se utilizan como apoyo al método.
¿Cuáles son los atributos de un buen software?	El software debe tener la funcionalidad y el rendimiento requeridos por el usuario, además de ser mantenible, confiable y fácil de utilizar.
¿Cuáles con los retos fundamentales a los que se enfrenta la ingeniería de software?	Enfrentarse con la creciente diversidad, las demandas para reducir los tiempos de entrega y el desarrollo de software fiable.

Figura 1.1 Preguntas frecuentes sobre la ingeniería del software.

tar estos programas, un sistema de documentación que describe la estructura del sistema, la documentación para el usuario que explica cómo utilizar el sistema y sitios web que permitan a los usuarios descargar la información de productos recientes.

Los ingenieros de software se concentran en el desarrollo de productos de software, es decir, software que se vende a un cliente. Existen dos tipos de productos de software:

1. *Productos genéricos.* Son sistemas aislados producidos por una organización de desarrollo y que se venden al mercado abierto a cualquier cliente que le sea posible comprarlos. Ejemplos de este tipo de producto son el software para PCs tales como bases de datos, procesadores de texto, paquetes de dibujo y herramientas de gestión de proyectos.
2. *Productos personalizados (o hechos a medida).* Son sistemas requeridos por un cliente en particular. Un contratista de software desarrolla el software especialmente para ese cliente. Ejemplos de este tipo de software son los sistemas de control para instrumentos electrónicos, sistemas desarrollados para llevar a cabo procesos de negocios específicos y sistemas de control del tráfico aéreo.

Una diferencia importante entre estos diferentes tipos de software es que, en los productos genéricos, la organización que desarrolla el software controla su especificación. La especificación de los productos personalizados, por lo general, es desarrollada y controlada por la organización que compra el software. Los desarrolladores de software deben trabajar con esa especificación.

No obstante, la línea de separación entre estos tipos de productos se está haciendo cada vez más borrosa. Cada vez más compañías de software empiezan con un sistema genérico y lo adaptan a las necesidades de un cliente en particular. Los sistemas de planificación de recursos empresariales (ERP), como los sistemas SAP, son el mejor ejemplo de este enfoque. Aquí, un sistema largo y complejo se adapta a una compañía incorporando información sobre reglas de negocio y de procesos, informes, etcétera.

1.1.2 ¿Qué es la ingeniería del software?

La ingeniería del software es una disciplina de la ingeniería que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento de éste después de que se utiliza. En esta definición, existen dos frases clave:

1. *Disciplina de la ingeniería.* Los ingenieros hacen que las cosas funcionen. Aplican teorías, métodos y herramientas donde sean convenientes, pero las utilizan de forma selectiva y siempre tratando de descubrir soluciones a los problemas, aun cuando no existan teorías y métodos aplicables para resolverlos. Los ingenieros también saben que deben trabajar con restricciones financieras y organizacionales, por lo que buscan soluciones tomando en cuenta estas restricciones.
2. *Todos los aspectos de producción de software.* La ingeniería del software no sólo comprende los procesos técnicos del desarrollo de software, sino también con actividades tales como la gestión de proyectos de software y el desarrollo de herramientas, métodos y teorías de apoyo a la producción de software.

En general, los ingenieros de software adoptan un enfoque sistemático y organizado en su trabajo, ya que es la forma más efectiva de producir software de alta calidad. Sin embargo, aunque la ingeniería consiste en seleccionar el método más apropiado para un conjunto de circunstancias, un enfoque más informal y creativo de desarrollo podría ser efectivo en al-

gunas circunstancias. El desarrollo informal es apropiado para el desarrollo de sistemas basados en Web, los cuales requieren una mezcla de técnicas de software y de diseño gráfico.

1.1.3 ¿Cuál es la diferencia entre ingeniería del software y ciencia de la computación?

Esencialmente, la ciencia de la computación se refiere a las teorías y métodos subyacentes a las computadoras y los sistemas de software, mientras que la ingeniería del software se refiere a los problemas prácticos de producir software. Los ingenieros de software requieren ciertos conocimientos de ciencia de la computación, de la misma forma que los ingenieros eléctricos requieren conocimientos de física.

Lo ideal sería que todos los ingenieros de software conocieran las teorías de la ciencia de la computación, pero en realidad éste no es el caso. Los ingenieros de software a menudo utilizan enfoques *ad hoc* para desarrollar el software. Las ingeniosas teorías de la ciencia de la computación no siempre pueden aplicarse a problemas reales y complejos que requieren una solución de software.

1.1.4 ¿Cuál es la diferencia entre ingeniería del software e ingeniería de sistemas?

La ingeniería de sistemas se refiere a todos los aspectos del desarrollo y de la evolución de sistemas complejos donde el software desempeña un papel principal. Por lo tanto, la ingeniería de sistemas comprende el desarrollo de hardware, políticas y procesos de diseño y distribución de sistemas, así como la ingeniería del software. Los ingenieros de sistemas están involucrados en la especificación del sistema, en la definición de su arquitectura y en la integración de las diferentes partes para crear el sistema final. Están menos relacionados con la ingeniería de los componentes del sistema (hardware, software, etc.).

La ingeniería de sistemas es más antigua que la del software. Por más de 100 años, las personas han especificado y construido sistemas industriales complejos, como aviones y plantas químicas. Sin embargo, puesto que se ha incrementado el porcentaje de software en los sistemas, las técnicas de ingeniería del software tales como el modelado de casos de uso y la gestión de la configuración se utilizan en el proceso de ingeniería de sistemas. En el Capítulo 2 se trata con mayor detalle la ingeniería de sistemas.

1.1.5 ¿Qué es un proceso del software?

Un proceso del software es un conjunto de actividades y resultados asociados que producen un producto de software. Estas actividades son llevadas a cabo por los ingenieros de software. Existen cuatro actividades fundamentales de procesos (incluidas más adelante en este libro) que son comunes para todos los procesos del software. Estas actividades son:

1. *Especificación del software* donde los clientes e ingenieros definen el software a producir y las restricciones sobre su operación.
2. *Desarrollo del software* donde el software se diseña y programa.
3. *Validación del software* donde el software se valida para asegurar que es lo que el cliente requiere.
4. *Evolución del software* donde el software se modifica para adaptarlo a los cambios requeridos por el cliente y el mercado.

Diferentes tipos de sistemas necesitan diferentes procesos de desarrollo. Por ejemplo, el software de tiempo real en un avión tiene que ser completamente especificado antes de que empiece el desarrollo, mientras que en un sistema de comercio electrónico, la especificación y el programa normalmente son desarrollados juntos. Por lo tanto, estas actividades genéricas pueden organizarse de diferentes formas y describirse en diferentes niveles de detalle para diferentes tipos de software. Sin embargo, el uso de un proceso inadecuado del software puede reducir la calidad o la utilidad del producto de software que se va a desarrollar y/o incrementar los costes de desarrollo.

En el Capítulo 4 se tratan con más detalle los procesos del software, y en el Capítulo 28 se aborda el tema de la mejora de dicho proceso.

1.1.6 ¿Qué es un modelo de procesos del software?

Un modelo de procesos del software es una descripción simplificada de un proceso del software que presenta una visión de ese proceso. Estos modelos pueden incluir actividades que son parte de los procesos y productos de software y el papel de las personas involucradas en la ingeniería del software. Algunos ejemplos de estos tipos de modelos que se pueden producir son:

1. *Un modelo de flujo de trabajo.* Muestra la secuencia de actividades en el proceso junto con sus entradas, salidas y dependencias. Las actividades en este modelo representan acciones humanas.
2. *Un modelo de flujo de datos o de actividad.* Representa el proceso como un conjunto de actividades, cada una de las cuales realiza alguna transformación en los datos. Muestra cómo la entrada en el proceso, tal como una especificación, se transforma en una salida, tal como un diseño. Pueden representar transformaciones llevadas a cabo por las personas o por las computadoras.
3. *Un modelo de roll/acción.* Representa los roles de las personas involucrada en el proceso del software y las actividades de las que son responsables.

La mayor parte de los modelos de procesos del software se basan en uno de los tres modelos generales o paradigmas de desarrollo de software:

1. *El enfoque en cascada.* Considera las actividades anteriores y las representa como fases de procesos separados, tales como la especificación de requerimientos, el diseño del software, la implementación, las pruebas, etcétera. Después de que cada etapa queda definida «se firma» y el desarrollo continúa con la siguiente etapa.
2. *Desarrollo iterativo.* Este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones muy abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga las necesidades de dicho cliente. El sistema puede entonces ser entregado. De forma alternativa, se puede reimplementar utilizando un enfoque más estructurado para producir un sistema más sólido y mantenible.
3. *Ingeniería del software basada en componentes (CBSE).* Esta técnica supone que las partes del sistema existen. El proceso de desarrollo del sistema se enfoca en la integración de estas partes más que desarrollarlas desde el principio. En el Capítulo 19 se estudia la CBSE.

En los Capítulos 4 y 17 se tratarán nuevamente estos modelos de procesos genéricos.

1.1.7 ¿Cuáles son los costos de la ingeniería del software?

No existe una respuesta sencilla a esta pregunta ya que la distribución de costos a través de las diferentes actividades en el proceso del software depende del proceso utilizado y del tipo de software que se vaya a desarrollar. Por ejemplo, el software de tiempo real normalmente requiere una validación y pruebas más extensas que los sistemas basados en web. Sin embargo, cada uno de los diferentes enfoques genéricos al desarrollo del software tiene un perfil de distribución de costos diferente a través de las actividades del proceso del software. Si se considera que el costo total del desarrollo de un sistema de software complejo es de 100 unidades de costo, la Figura 1.2 muestra cómo se gastan éstas en las diferentes actividades del proceso.

En el enfoque en cascada, los costos de especificación, diseño, implementación e integración se miden de forma separada. Observe que la integración y pruebas del sistemas son las actividades de desarrollo más caras. Normalmente, éste supone alrededor del 40% del costo del desarrollo total, pero para algunos sistemas críticos es probable que sea al menos el 50% de los costos de desarrollo del sistema.

Si el software se desarrolla utilizando un enfoque iterativo, no existe división entre la especificación, el diseño y el desarrollo. En este enfoque, los costos de la especificación se reducen debido a que sólo se produce la especificación de alto nivel antes que el desarrollo. La especificación, el diseño, la implementación, la integración y las pruebas se llevan a cabo en paralelo dentro de una actividad de desarrollo. Sin embargo, aún se necesita una actividad independiente de pruebas del sistema una vez que la implementación inicial esté completa.

La ingeniería del software basada en componentes sólo ha sido ampliamente utilizada durante un corto periodo de tiempo. En este enfoque, no tenemos figuras exactas para los costos de las diferentes actividades del desarrollo de software. Sin embargo, sabemos que los

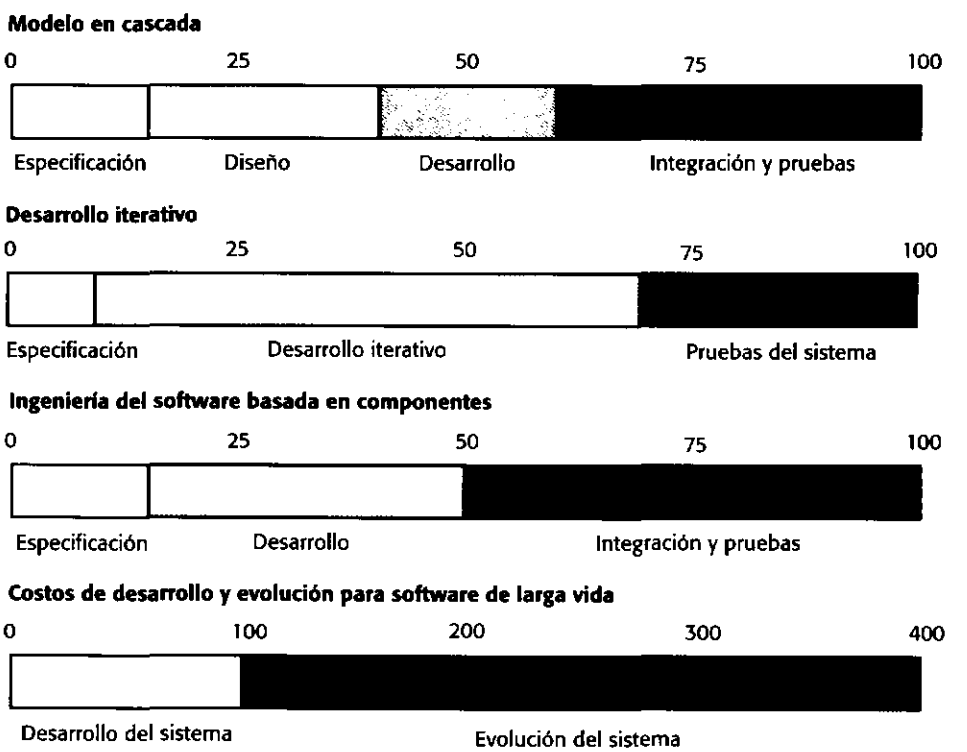


Figura 1.2 Distribución de costos de las actividades de la ingeniería del software.

costos de desarrollo se reducen en relación a los costos de integración y pruebas. Los costos de integración y pruebas se incrementan porque tenemos que asegurarnos de que los componentes que utilizamos cumplen realmente su especificación y funcionan como se espera con otros componentes.

Además de los costos de desarrollo, existen costos asociados a cambios que se le hacen al software una vez que está en uso. Los costos de evolución varían drásticamente dependiendo del tipo de sistema. Para sistemas software de larga vida, tales como sistemas de orden y de control que pueden ser usados durante 10 años o más, estos costos exceden a los de desarrollo por un factor de 3 o 4, como se muestra en la barra inferior en la Figura 1.3. Sin embargo, los sistemas de negocio más pequeños tienen una vida mucho más corta y, correspondientemente, costos de evolución más reducidos.

Esta distribución de costos se cumple para el software personalizado, el cual es especificado por un cliente y desarrollado por un contratista. Para productos de software que se venden (mayormente) para PCs, el perfil del costo es diferente. Estos productos comúnmente se desarrollan a partir de una especificación inicial utilizando un enfoque de desarrollo evolutivo. Los costos de la especificación son relativamente bajos. Sin embargo, debido que se pretende utilizarlos en diferentes configuraciones, deben ser probados a fondo. La Figura 1.3 muestra el perfil del costo que se puede esperar para estos productos.

Los costos de evolución para productos de software genéricos son difíciles de estimar. En muchos casos, existe poca evolución de un producto. Una vez que una versión de producto se entrega, se inicia el trabajo para entregar la siguiente y, por razones de mercadotecnia, ésta se presenta como un producto nuevo (pero compatible) más que como una versión modificada de un producto que el usuario ya adquirió. Por lo tanto, los costos de evolución no se consideran de forma separada como en el caso del software personalizado, sino que son sencillamente los costos del desarrollo para la siguiente versión del sistema.

1.1.8 ¿Qué son los métodos de la ingeniería del software?

Un método de ingeniería del software es un enfoque estructurado para el desarrollo de software cuyo propósito es facilitar la producción de software de alta calidad de una forma costeable. Métodos como Análisis Estructurado (DeMarco, 1978) y JSD (Jackson, 1983) fueron los primeros desarrollados en los años 70. Estos métodos intentaron identificar los componentes funcionales básicos de un sistema, de tal forma que los métodos orientados a funciones aún se utilizan ampliamente. En los años 80 y 90, estos métodos orientados a funciones fueron complementados por métodos orientados a objetos, como los propuestos por Booch (1994) y Rumbaugh (Rumbaugh *et al.*, 1991). Estos diferentes enfoques se han integrado en un solo enfoque unificado basado en el Lenguaje de Modelado Unificado (UML) (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a; Rumbaugh *et al.*, 1999b).

No existe un método ideal, y métodos diferentes tienen distintas áreas donde son aplicables. Por ejemplo, los métodos orientados a objetos a menudo son apropiados para sistemas interactivos, pero no para sistemas con requerimientos rigurosos de tiempo real.

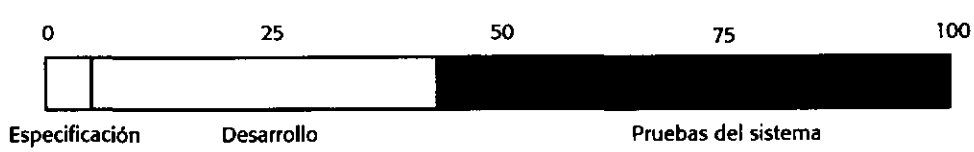


Figura 1.3 Costos del desarrollo del producto.

Todos los métodos se basan en la idea de modelos gráficos de desarrollo de un sistema y en el uso de estos modelos como un sistema de especificación o diseño. Los métodos incluyen varios componentes diferentes (Figura 1.4).

1.1.9 ¿Qué es CASE?

CASE (Ingeniería del Software Asistida por Computadora) comprende un amplio abanico de diferentes tipos de programas que se utilizan para ayudar a las actividades del proceso del software, como el análisis de requerimientos, el modelado de sistemas, la depuración y las pruebas. En la actualidad, todos los métodos vienen con tecnología CASE asociada, como los editores para las notaciones utilizadas en el método, módulos de análisis que verifican el modelo del sistema según las reglas del método y generadores de informes que ayudan a crear la documentación del sistema. Las herramientas CASE también incluyen un generador de código que automáticamente genera código fuente a partir del modelo del sistema y de algunas guías de procesos para los ingenieros de software.

1.1.10 ¿Cuáles son los atributos de un buen software?

Así como los servicios que proveen, los productos de software tienen un cierto número de atributos asociados que reflejan la calidad de ese software. Estos atributos no están directamente asociados con lo que el software hace. Más bien, reflejan su comportamiento durante su ejecución y en la estructura y organización del programa fuente y en la documentación asociada. Ejemplos de estos atributos (algunas veces llamados atributos no funcionales) son el tiempo de respuesta del software a una pregunta del usuario y la comprensión del programa fuente.

El conjunto específico de atributos que se espera de un sistema de software depende obviamente de su aplicación. Por lo tanto, un sistema bancario debe ser seguro, un juego interactivo debe tener capacidad de respuesta, un interruptor de un sistema telefónico debe ser fiable, etcétera. Esto se generaliza en el conjunto de atributos que se muestra en la Figura 1.5, el cual tiene las características esenciales de un sistema de software bien diseñado.

Descripciones del modelo del sistema	**Descripciones de los modelos del sistema que desarrollará y la notación utilizada para definir estos modelos.**	**Modelos de objetos, de flujo de datos, de máquina de estado, etcétera.**
Reglas	**Restricciones que siempre aplican a los modelos de sistemas.**	**Cada entidad de un modelo de sistema debe tener un nombre único.**
Recomendaciones	**Heurística que caracteriza una buena práctica de diseño en este método. Seguir estas recomendaciones debe dar como resultado un modelo del sistema bien organizado.**	**Ningún objeto debe tener más de siete subobjetos asociados a él.**
Guías en el proceso	**Descripciones de las actividades que deben seguirse para desarrollar los modelos del sistema y la organización de estas actividades.**	**Los atributos de los objetos deben documentarse antes de definir las operaciones asociadas a un objeto.**

Figura 1.4 Componentes del método.

Mantenibilidad	**El software debe escribirse de tal forma que pueda evolucionar para cumplir las necesidades de cambio de los clientes. Éste es un atributo crítico debido a que el cambio en el software es una consecuencia inevitable de un cambio en el entorno de negocios.**
Confiabilidad	**La confiabilidad del software tiene un gran número de características, incluyendo la fiabilidad, protección y seguridad. El software confiable no debe causar daños físicos o económicos en el caso de una falla del sistema.**
Eficiencia	**El software no debe hacer que se malgasten los recursos del sistema, como la memoria y los ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, utilización de la memoria, etcétera.**
Usabilidad	**El software debe ser fácil de utilizar, sin esfuerzo adicional, por el usuario para quien está diseñado. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.**

Figura 1.5 Atributos esenciales de un buen software.

1.1.11 ¿Cuáles son los retos fundamentales que afronta la ingeniería del software?

En el siglo XXI, la ingeniería del software afronta tres retos fundamentales:

1. *El reto de la heterogeneidad.* Cada vez más, se requiere que los sistemas operen como sistemas distribuidos en redes que incluyen diferentes tipos de computadoras y con diferentes clases de sistemas de soporte. A menudo es necesario integrar software nuevo con sistemas heredados más viejos escritos en diferentes lenguajes de programación. El reto de la heterogeneidad es desarrollar técnicas para construir software confiable que sea lo suficientemente flexible para adecuarse a esta heterogeneidad.
2. *El reto de la entrega.* Muchas técnicas tradicionales de ingeniería del software consumen tiempo. El tiempo que éstas consumen es para producir un software de calidad. Sin embargo, los negocios de hoy en día deben tener una gran capacidad de respuesta y cambiar con mucha rapidez. Su software de soporte también debe cambiar con la misma rapidez. El reto de la entrega es reducir los tiempos de entrega para sistemas grandes y complejos sin comprometer la calidad del sistema.
3. *El reto de la confianza.* Puesto que el software tiene relación con todos los aspectos de nuestra vida, es esencial que podamos confiar en él. Esto es especialmente importante en sistemas remotos de software a los que se accede a través de páginas web o de interfaces de servicios web. El reto de la confianza es desarrollar técnicas que demuestren que los usuarios pueden confiar en el software.

Por supuesto, éstos no son independientes. Por ejemplo, es necesario hacer cambios rápidos a los sistemas heredados para proveerlos de una interfaz de servicio web. Para tratar estos retos, necesitaremos nuevas herramientas y técnicas, así como formas innovadoras de combinación y uso de métodos de ingeniería del software existentes.

1.2 Responsabilidad profesional y ética

Como otras disciplinas de la ingeniería, la ingeniería del software se lleva a cabo dentro de un marco legal y social que limita la libertad de los ingenieros. Los ingenieros de software

deben aceptar que su trabajo comprende responsabilidades más amplias que simplemente la aplicación de habilidades técnicas. Deben comportarse de una forma ética y moral responsable si es que desean ser respetados como profesionales.

No basta con decir que usted siempre debe poseer estándares normales de honestidad e integridad. No debería utilizar su capacidad y sus habilidades para comportarse de forma deshonesta o de forma que deshonre la profesión de la ingeniería del software. Sin embargo, existen áreas donde los estándares de comportamiento aceptable no están acotados por las leyes, sino por la más tenue noción de responsabilidad profesional. Algunas de éstas son:

1. *Confidencialidad.* Usted normalmente debe respetar la confidencialidad de sus empleadores o clientes independientemente de que se haya firmado un acuerdo formal de confidencialidad.
2. *Competencia.* No debe falsificar su nivel de competencia, ni aceptar conscientemente trabajos que están fuera de su capacidad.
3. *Derechos de propiedad intelectual.* Debe ser consciente de las leyes locales que gobiernan el uso de la propiedad intelectual, como las patentes y el copyright. Debe asegurarse de que la propiedad intelectual de los empleadores y clientes está protegida.
4. *Uso inapropiado de los computadoras.* No debe emplear sus habilidades técnicas para utilizar de forma inapropiada las computadoras de otras personas. El uso inapropiado de las computadoras va desde los relativamente triviales (utilizar juegos en la máquina de un empleado, por ejemplo) hasta los extremadamente serios (difusión de virus).

Las sociedades e instituciones profesionales tiene que desempeñar un papel importante en el establecimiento de estándares éticos. Organizaciones como la ACM, el IEEE (Instituto de Ingenieros Eléctricos y Electrónicos) y la British Computer Society publican un código de conducta profesional o de ética. Los miembros de estas organizaciones se comprometen a cumplir ese código cuando se inscriben en ellas. Estos códigos de conducta generalmente se refieren al comportamiento ético fundamental.

La ACM y el IEEE han cooperado para crear un código de ética y práctica profesional. Este código existe en forma reducida, como se muestra en la Figura 1.6, y en forma extendida (Gotterbarn *et al.*, 1999), la cual agrega detalle y sustancia a la versión reducida. El fundamento sobre el que se asienta este código se resume en los dos primeros párrafos de la versión extendida:

> *Las computadoras tienen un papel central y creciente en el comercio, la industria, el gobierno, la medicina, la educación, el entretenimiento y la sociedad en general. Los ingenieros de software son aquellos que contribuyen con su participación directa o por su enseñanza, al análisis, especificación, diseño, desarrollo, certificación, mantenimiento y pruebas de los sistemas de software. Debido a sus papeles en el desarrollo de sistemas de software, los ingenieros de software tienen significativas oportunidades de hacer el bien o causar daño, permitir que otros hagan el bien o causen daño, o influir en otros para hacer el bien o causar daño. Para asegurar, tanto como sea posible, que sus esfuerzos serán utilizados para bien, los ingenieros de software deben comprometerse consigo mismos para hacer de la ingeniería del software una profesión de beneficio y respeto. De acuerdo con este compromiso, los ingenieros de software deben cumplir el siguiente Código de Ética y Práctica Profesional.*

Código de Ética y Práctica Profesional de la Ingeniería del Software

ACM/IEEE-CS Fuerza de Tarea Conjunta sobre Ética y Práctica Profesional en la Ingeniería del Software

PREÁMBULO

La versión corta del código resume las aspiraciones en un alto nivel de abstracción. Las cláusulas que se incluyen en la versión completa proporcionan ejemplos y detalles de cómo estas aspiraciones cambian la forma en que actuamos como profesionales de la ingeniería del software. Sin las aspiraciones, los detalles pueden llegar a ser sólo términos jurídicos tediosos; sin los detalles, las aspiraciones pueden llegar a ser altisonantes pero vacías; juntas, las aspiraciones y los detalles forman un código cohesivo.

Los ingenieros de software deben comprometerse consigo mismos para hacer del análisis, la especificación, el diseño, el desarrollo, las pruebas y el mantenimiento del software una profesión beneficiosa y respetada. En concordancia con su compromiso con la salud, la seguridad y el bienestar del público, los ingenieros de software deben adherirse a los siguientes ocho principios:

1. **PÚBLICO – Los ingenieros de software deberán actuar en consonancia con el interés público.**
2. **CLIENTE Y EMPLEADOR – Los ingenieros de software deberán actuar de forma que respondan a los intereses de sus clientes y empleadores siendo consecuentes con el interés público.**
3. **PRODUCTO – Los ingenieros de software deberán asegurar que sus productos y las modificaciones asociadas cumplan los más altos estándares profesionales posibles.**
4. **JUICIO – Los ingenieros de software deberán mantener la integridad e independencia en sus juicios profesionales.**
5. **GESTIÓN – Los gerentes y líderes ingenieros de software deberán suscribir y promocionar un enfoque ético en la gestión del desarrollo y mantenimiento del software.**
6. **PROFESIÓN – Los ingenieros de software deberán mantener la integridad y reputación de la profesión de acuerdo con el interés público.**
7. **COLEGAS – Los ingenieros de software deberán ser imparciales y apoyar a sus colegas.**
8. **PERSONAL – Durante toda sus existencia, los ingenieros de software deberán aprender lo concerniente a la práctica de su profesión y promocionar un enfoque ético en la práctica de su profesión.**

Figura 1.6 Código de Ética de la ACM/IEEE (©IEEE/ACM 1999).

El código contiene ocho principios relacionados con el comportamiento y con las decisiones hechas por ingenieros de software profesionales, incluyendo practicantes, educadores, administradores, supervisores y creadores de políticas, así como aprendices y estudiantes de la profesión. Los principios identifican las relaciones éticas en las que los individuos, grupos y organizaciones participan, y las obligaciones primarias dentro de estas relaciones. Las cláusulas de cada principio son ilustraciones de algunas de las obligaciones incluidas en estas relaciones. Estas obligaciones se fundamentan en la humanidad del ingeniero de software, con especial cuidado en la gente afectada por el trabajo de los ingenieros de software, y los elementos únicos de la práctica de la ingeniería del software. El código prescribe éstas como obligaciones de cualquiera que se llame o que aspire a ser un ingeniero de software.

En cualquier situación en la que diferentes personas tienen distintos puntos de vista y objetivos, es posible encontrar problemas éticos. Por ejemplo, si usted está en desacuerdo, en principio, con las políticas de un directivo de categoría superior en la compañía, ¿cómo debería reaccionar? Desde luego, esto depende de cada individuo y de la naturaleza de la discordancia. ¿Es mejor argumentar a favor de su posición dentro de la organización o renunciar de acuerdo con sus principios? Si piensa que existen problemas con un proyecto de software, ¿cuándo se deben comunicar éstos al gerente? Si éstos se discuten cuando son sólo

una sospecha, puede ser una sobre reacción a la situación, si lo deja para más tarde, puede ser imposible resolver las dificultades.

Tales problemas éticos aparecen en nuestra vida profesional y, afortunadamente, en muchos casos son relativamente menores o se pueden resolver sin mucha dificultad. Cuando no se puedan resolver, los ingenieros se enfrentarán, quizás, con otro problema. La acción con base en sus principios podría ser renunciar a su trabajo, pero esto puede afectar a otros, por ejemplo, a sus colaboradores o sus hijos.

Una situación particularmente difícil para los ingenieros profesionales surge cuando su empleador actúa de una forma no ética. Por ejemplo, una compañía es responsable de desarrollar un sistema crítico de seguridad y, debido a las presiones de tiempo, falsifica la validación de protección de los registros. ¿Es responsabilidad del ingeniero mantener la confidencialidad o alertar al cliente o hacer público, de alguna forma, que el sistema entregado es inseguro?

El problema aquí es que no existen absolutos cuando se trata de protección. Aun cuando el sistema no haya sido validado de acuerdo con los criterios predefinidos, éstos pueden ser demasiado estrictos. El sistema puede, de hecho, operar de forma segura a través de su tiempo de vida. Pero puede darse el caso de que, aun cuando sea validado apropiadamente, el sistema falle y cause un accidente. El descubrimiento temprano de problemas puede resultar perjudicial para el empleador y otros empleados; la falta de descubrimiento de problemas puede resultar perjudicial para otros.

Debe tomar su propia decisión en estos temas. La posición ética apropiada depende enteramente del punto de vista de los individuos que están involucrados. En este caso, el potencial para el daño, el grado del daño y la gente afectada por él deben influir en la decisión. Si la situación es muy peligrosa, se justifica su publicación en la prensa nacional (por ejemplo). Sin embargo, se debe tratar de resolver la situación respetando los derechos del empleador.

Otra cuestión ética es la participación en el desarrollo de sistemas militares y nucleares. Algunas personas tienen una opinión firme sobre estos temas y no desean participar en ningún desarrollo de sistemas asociados con sistemas militares. Otros trabajarán en sistemas militares, pero no en sistemas de armamento. Algunos otros sentirán que la defensa de la nación es un principio fundamental y no tienen objeciones éticas para trabajar en sistemas de armamento.

En esta situación, es importante que tanto empleadores como empleados se hagan saber con anticipación sus puntos de vista. Cuando una organización está relacionada con el trabajo militar o nuclear, le debe ser posible especificar si los empleados pueden aceptar cualquier tarea. De igual forma, si un empleado hace patente que no desea trabajar en tales sistemas, los empleadores no deben presionarlo posteriormente.

El área de ética y responsabilidad profesional ha recibido creciente atención en los pasados años. Los principios de ética se pueden considerar desde un punto de vista filosófico, y la ética de la ingeniería del software se debe tratar con referencia a estos principios básicos. Éste es el enfoque considerado por Laudon (Laudon, 1995) y, de forma menos extensa, por Huff y Martin (Huff y Martin, 1995).

Sin embargo, este enfoque me resulta abstracto y difícil de relacionar con mi experiencia diaria. Prefiero un enfoque más concreto comprendido en códigos de conducta y práctica. Creo que la ética se analiza mejor en un contexto de ingeniería del software y no como un tema por sí solo. En este libro, por lo tanto, no incluiré consideraciones éticas abstractas sino que, donde sea apropiado, incluiré ejemplos en los ejercicios que puedan servir de punto de partida para una discusión ética.

PUNTOS CLAVE

- **La ingeniería del software es una disciplina de ingeniería que comprende todos los aspectos de la producción de software.**
- **Los productos de software consisten en programas desarrollados y en la documentación asociada. Los atributos esenciales de los productos son la mantenibilidad, confiabilidad, eficiencia y aceptabilidad.**
- **El proceso del software incluye todas las actividades relativas al desarrollo del software. Las actividades de alto nivel de especificación del software, el desarrollo, la validación y la evolución son parte de todos los procesos software.**
- **Los métodos son formas organizadas de producir software. Incluyen sugerencias para el proceso que se debe seguir, la notación que se va a utilizar, los modelos del sistema que hay que desarrollar y las reglas que gobiernan estos modelos y las pautas de diseño.**
- **Las herramientas CASE son sistemas de software que están diseñados para ayudar a las actividades rutinarias del proceso del software, como editar diagramas de diseño, verificar la consistencia de éstos y mantener un banco de pruebas de los programas ejecutados.**
- **Los ingenieros de software tienen responsabilidades en la profesión de la ingeniería y en la sociedad. No sólo deben estar pendientes de los aspectos técnicos.**
- **Las sociedades profesionales publican códigos de conducta que definen los estándares de comportamiento esperado por sus miembros.**

LECTURAS ADICIONALES

Fundamentals of Software Engineering. Un texto general sobre la ingeniería del software que da una perspectiva de la materia bastante diferente de la que da este libro. (C. Ghezi *et. al.*, Prentice Hall, 2003.)

«Software engineering: The state of the practice». Un número especial del *IEEE Software* que incluye varios artículos que estudian la práctica actual en la ingeniería del software, cómo ha cambiado y el grado en el cual se utilizan nuevas tecnologías del software. [*IEEE Software,* 20 (6), noviembre de 2003.]

Software Engineering: An Engineering Approach. Un texto general que toma un enfoque bastante diferente del de mi libro pero que incluye algunos casos de estudio útiles. (J. F. Peters y W. Pedrycz, 2000, John Wiley & Sons.)

Professional Issues in Software Engineering. Éste es un excelente libro que analiza los aspectos legales, profesionales y éticos. Yo prefiero su enfoque práctico a los textos más teóricos sobre ética. (F. Bott *et al.*, 3.ª edición, 2000, Taylor & Francis.)

«Software Engineering Code of Ethics is approved». Un artículo que estudia las bases del desarrollo del Código de Ética de la ACM/IEEE e incluye tanto la versión reducida como la versión extendida del código. (*Comm. ACM*, D. Gotterbarn *et al.*, octubre de 1999.)

«No silver bullet: Essence and accidents of software engineering». Pese al tiempo transcurrido desde su publicación, este artículo es una buena introducción general a los problemas de la ingeniería del software. El mensaje esencial del artículo, que no hay una sola respuesta a los problemas de la ingeniería del software, no ha cambiado. [F. P. Brooks, *IEEE Computer,* 20(4), abril de 1987.]

EJERCICIOS

1.1 Haciendo referencia a la distribución de costos del software inidicados en la Sección 1.1.6, explique por qué es apropiado considerar que el software es más que programas que son ejecutados por los usuarios finales de un sistema.

1.2 ¿Cuáles son las diferencias entre el desarrollo de un producto de software genérico y el desarrollo de un software personalizado?

1.3 ¿Cuáles son los cuatro atributos importantes que todos los productos de software deben tener? Sugiera otros cuatro atributos que pueden ser significativos.

1.4 ¿Cuál es la diferencia entre un modelo del proceso del software y un proceso del software? Sugiera dos formas en las que un modelo del proceso del software ayuda en la identificación de posibles mejoras del proceso.

1.5 Explique por qué los costos de pruebas de software son particularmente altos para productos de software genéricos que se venden a un mercado amplio.

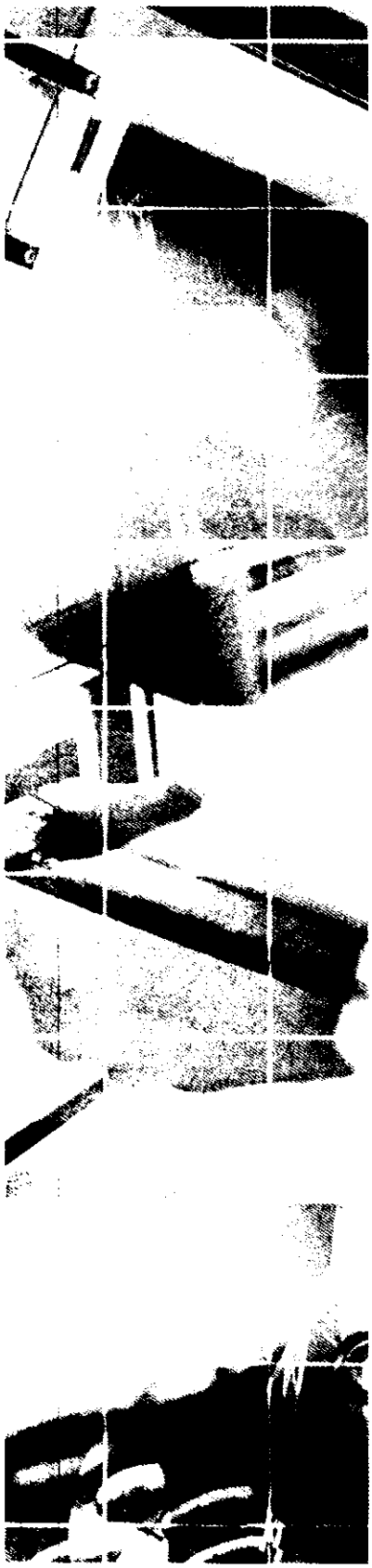
1.6 Los métodos de la ingeniería del software se empezaron a utilizar cuando la tecnología CASE estuvo disponible para apoyarlos. Mencione cinco tipos de métodos de ayuda que proporcionen las herramientas CASE.

1.7 Además de los retos de la heterogeneidad, la rápida entrega y la confianza, identifique otros problemas y retos que la ingeniería del software afrontará en el siglo XXI.

1.8 Comente si los ingenieros profesionales deben atestiguar de la misma forma que los doctores o los abogados.

1.9 Para cada una de las cláusulas del Código de Ética de la ACM/IEEE que se muestra en la Figura 1.6, sugiera un ejemplo apropiado que ilustre esa cláusula.

1.10 Para contrarrestar al terrorismo, muchos países están planeando el desarrollo de sistemas informáticos que sigan la pista de un gran número de sus ciudadanos y de sus acciones. Desde luego, esto tiene implicaciones sobre la privacidad. Comente la ética de desarrollar este tipo de sistema.



2 Sistemas socio-técnicos

Objetivos

Los objetivos de este capítulo son introducir el concepto de un sistema socio-técnico —que incluye personas, software y hardware— y el proceso de la ingeniería de sistemas. Cuando haya leído este capítulo:

- sabrá qué se entiende por sistema socio-técnico y comprenderá la diferencia entre un sistema técnico informático y un sistema socio-técnico;
- conocerá el concepto de propiedades emergentes de los sistemas, como la fiabilidad, el rendimiento, la seguridad y la protección;
- entenderá las actividades implicadas en el proceso de la ingeniería de sistemas;
- entenderá por qué el contexto organizacional de un sistema afecta a su diseño y uso;
- sabrá qué significa «sistema heredado», y por qué estos sistemas son a menudo críticos para las operaciones de muchos negocios.

Contenidos

El término *sistema* es universalmente usado. Hablamos sobre sistemas informáticos, sistemas operativos, sistemas de pago, el sistema educacional, el sistema de gobierno, etcétera. Éstos son obviamente usos bastante diferentes de la palabra *sistema* aunque coinciden en que, de algún modo, el sistema es más que simplemente la suma de sus partes.

Sistemas muy abstractos tales como el sistema de gobierno están fuera del ámbito de este libro. Consecuentemente, me centro aquí en sistemas que incluyen computadoras y que tienen algún propósito específico, como permitir la comunicación, ayudar a la navegación y calcular salarios. Por lo tanto, una definición útil de estos tipos de sistemas es la siguiente:

> *Un sistema es una colección de componentes interrelacionados que trabajan conjuntamente para cumplir algún objetivo.*

Esta definición general comprende una amplia serie de sistemas. Por ejemplo, un sistema tan simple como un bolígrafo incluye tres o cuatro componentes hardware. En contraste, un sistema de control del tráfico aéreo incluye miles de componentes hardware y software además de los usuarios humanos que toman decisiones basadas en la información del sistema.

Los sistemas que incluyen software se dividen en dos categorías:

- *Sistemas técnicos informáticos:* son sistemas que incluyen componentes hardware y software, pero no procedimientos y procesos. Ejemplos de sistemas técnicos son las televisiones, los teléfonos móviles y la mayoría del software de las computadoras personales. Los individuos y organizaciones usan sistemas técnicos para algún fin, pero el conocimiento de este fin no es parte del sistema. Por ejemplo, el procesador de textos que estoy utilizando no es consciente de que se está utilizando para escribir un libro.
- *Sistemas socio-técnicos:* comprenden uno o más sistemas técnicos pero, crucialmente, también incluyen conocimiento de cómo debe usarse el sistema para alcanzar algún objetivo más amplio. Esto quiere decir que estos sistemas han definido los procesos operativos, incluyen personas (los operadores) como partes inherentes del sistema, son gobernados por políticas y reglas organizacionales y pueden verse afectados por restricciones externas tales como leyes nacionales y políticas reguladoras. Por ejemplo, este libro fue creado por un sistema socio-técnico de la industria editorial que incluye varios procesos y sistemas técnicos.

Las características esenciales de los sistemas socio-técnicos son las siguientes:

1. Tienen propiedades emergentes que son propiedades del sistema *como un todo* más que asociadas con partes individuales del sistema. Las propiedades emergentes dependen tanto de los componentes del sistema como de las relaciones entre ellos. Como esto es tan complejo, las propiedades emergentes sólo pueden ser evaluadas una vez que el sistema ha sido montado.
2. Son a menudo no deterministas. Esto significa que, cuando se presentan con una entrada específica, no siempre producen la misma salida. El comportamiento del sistema depende de operadores humanos, y las personas no siempre reaccionan de la misma forma. Además, el uso del sistema puede crear nuevas relaciones entre los componentes del sistema y, por lo tanto, cambiar su comportamiento emergente.
3. El grado en que el sistema apoya los objetivos organizacionales no sólo depende del sistema en sí mismo. También depende de la estabilidad de estos objetivos, de las relaciones y conflictos entre los objetivos organizacionales y de cómo las personas en la organización interpretan estos objetivos. Una nueva dirección puede reinterpretar los objetivos organizacionales para los que un sistema está diseñado, y un sistema «exitoso» puede convertirse en un «fracaso».

En este libro, se estudian los sistemas socio-técnicos que incluyen hardware y software, los cuales han definido procesos operativos y ofrecen una interfaz, implementada en software, a los usuarios humanos. Los ingenieros de software deben poseer un conocimiento de los sistemas socio-técnicos y la ingeniería de sistemas (White *et al.*, 1993; Thayer, 2002) debido a la importancia del software en estos sistemas. Por ejemplo, hubo menos de 10 megabytes de software en el programa espacial Apolo que puso un hombre en la Luna en 1969, pero existen más de 100 megabytes de software en los sistemas de control de la estación espacial Columbus.

Una característica de los sistemas es que las propiedades y el comportamiento de los componentes del sistema están inseparablemente entremezclados. El funcionamiento exitoso de cada componente del sistema depende del funcionamiento de otros componentes. Así, el software sólo puede funcionar si el procesador es operativo. El procesador sólo puede hacer cálculos si el sistema de software que define las operaciones se ha instalado de forma acertada.

Por lo general, los sistemas son jerárquicos y de este modo incluyen otros sistemas. Por ejemplo, un sistema de órdenes y control policiaco puede incluir un sistema de información geográfica para proporcionar los detalles de la localización de incidentes. Estos sistemas se denominan *subsistemas*. Una característica de éstos es que pueden operar por sí solos como sistemas independientes. Por lo tanto, el mismo sistema de información geográfica se puede utilizar en diferentes sistemas.

Puesto que el software es intrínsecamente flexible, los ingenieros de software deben resolver muchos problemas inesperados. Por ejemplo, digamos que la instalación de un radar se ha situado de tal forma que aparece una imagen fantasma de la imagen del radar. No es práctico mover el radar a un sitio con menos interferencias, por lo que los ingenieros de software tienen que encontrar otra técnica para eliminar estas imágenes fantasma. Su solución podría ser mejorar las capacidades del procesamiento de imágenes del software para eliminar las imágenes fantasma. Esto puede ralentizar el software de tal forma que el rendimiento sea inaceptable. El problema se puede entonces caracterizar como un «fallo de funcionamiento del software», mientras que, en realidad, fue un fallo en el proceso de diseño del sistema en su totalidad.

Esta situación, en que a los ingenieros de software se les deja el problema de mejorar las capacidades del software sin incrementar el costo del hardware, es muy común. Muchos de los llamados fallos de funcionamiento del software no son consecuencia de problemas inherentes a éste; son el resultado de tratar de cambiarlo para adecuarlo a las modificaciones en los requerimientos de la ingeniería de sistemas. Un buen ejemplo de esto es el fallo en el sistema de equipaje del aeropuerto de Denver (Swartz, 1996), donde se esperaba que el software de control se hiciera cargo de algunas limitaciones del equipo utilizado.

La ingeniería del software es, por lo tanto, crítica para el desarrollo acertado de complejos sistemas informáticos socio-técnicos. Como ingeniero de software, usted no debería ocuparse sólo del software en sí mismo, sino que además debería tener un conocimiento más amplio de cómo el software interactúa con otros sistemas hardware y software y cómo se debe usar. Este conocimiento le ayuda a entender los límites del software, a diseñar un mejor software y a participar como miembros iguales de un grupo de ingeniería de sistemas.

2.1 Propiedades emergentes de los sistemas

Las complejas relaciones entre los componentes de un sistema indican que el sistema es más que simplemente la suma de sus partes. Éste tiene propiedades que son propiedades del sistema como un todo. Estas *propiedades emergentes* (Checkland, 1981) no se pueden atribuir

a ninguna parte específica del sistema. Más bien, emergen sólo cuando los componentes del sistema han sido integrados. Algunas de estas propiedades pueden derivar directamente de las propiedades comparables de los subsistemas. Sin embargo, más a menudo, resultan de complejas interrelaciones de los subsistemas que no pueden, en la práctica, derivarse de las propiedades de los componentes individuales del sistema. En la Figura 2.1 se muestran ejemplos de algunas propiedades emergentes.

Existen dos tipos de propiedades emergentes:

1. *Las propiedades emergentes funcionales* aparecen cuando todas las partes de un sistema trabajan de forma conjunta para cumplir algún objetivo. Por ejemplo, una bicicleta tiene la propiedad funcional de ser un instrumento de transporte una vez que sus componentes se han conjuntado.
2. *Las propiedades emergentes no funcionales* se refieren al comportamiento de los sistemas en su entorno operativo. Ejemplos de propiedades no funcionales son la fiabilidad, el rendimiento, la seguridad y la protección. A menudo son factores críticos para sistemas informáticos, ya que un fallo mínimo en estas propiedades puede hacer inutilizable el sistema. Algunos usuarios puede que no necesiten ciertas funciones del sistema, por lo que éste puede ser aceptable sin ellas. Sin embargo, un sistema no fiable o demasiado lento es probablemente rechazado por todos los usuarios.

Para ilustrar la complejidad de las propiedades emergentes, considere la propiedad de la fiabilidad del sistema. La fiabilidad es un concepto complejo que siempre debe estudiarse en el nivel del sistema más que en el de los componentes individuales. Los componentes de un sistema son interdependientes, de tal forma que un fallo en uno de ellos se puede propagar a través del sistema y afectar a la operación de otros componentes. A veces es difícil predecir la manera en que las consecuencias de los fallos de los componentes se propagan a través del sistema. Por consiguiente, no se pueden hacer buenas estimaciones de la fiabilidad en conjunto del sistema de los datos de fiabilidad de los componentes del sistema.

Existen tres influencias conexas sobre la fiabilidad de un sistema:

1. *Fiabilidad del hardware.* ¿Cuál es la probabilidad de que un componente hardware falle y cuánto tiempo lleva reparar ese componente?

Volumen	**El volumen de un sistema (el espacio total ocupado) varía dependiendo de cómo estén ordenados y conectados los montajes de los componentes.**
Fiabilidad	**La fiabilidad del sistema depende de la fiabilidad de los componentes, pero interacciones inesperadas pueden causar nuevos tipos de fallos y, por lo tanto, afectar a la fiabilidad del sistema.**
Protección	**La protección del sistema (su capacidad para resistir ataques) es una propiedad compleja que no se puede medir fácilmente. Los ataques pueden ser ideados de forma que no fueron predichos por los diseñadores del sistema y así vencer las protecciones incorporadas.**
Reparabilidad	**Esta propiedad refleja hasta qué punto resulta fácil arreglar un problema con el sistema una vez que ha sido descubierto. Depende de la posibilidad de diagnosticar el problema, acceder a los componentes que son defectuosos y modificar o reemplazar estos componentes.**
Usabilidad	**Esta propiedad refleja cómo es de fácil usar el sistema. Depende de los componentes técnicos del sistema, sus operarios y su entorno de operaciones.**

Figura 2.1 Ejemplos de propiedades emergentes.

2. *Fiabilidad del software.* ¿Qué probabilidad hay de que un componente software produzca una salida incorrecta? Los fallos de funcionamiento del software normalmente son distintos de los del hardware en el sentido de que el software no se desgasta. Los fallos son normalmente transitorios por lo que el sistema puede continuar funcionando después de que se haya producido un resultado incorrecto.
3. *Fiabilidad del operador.* ¿Qué probabilidad existe de que un operador de un sistema cometa un error?

Estas influencias están fuertemente relacionadas. Los fallos de hardware pueden generar falsas señales fuera del rango de las entradas esperadas por el software. El software puede entonces comportarse de forma impredecible. Un error del operador es más probable en condiciones de tensión, como cuando ocurren fallos del sistema. Estos errores del operador pueden afectar al hardware, causando más fallos, y así sucesivamente. Por lo tanto, el fallo inicial recuperable puede convertirse rápidamente en un problema serio que requiera una parada completa del sistema.

Al igual que la fiabilidad, otras propiedades emergentes, como el rendimiento o la usabilidad, son difíciles de valorar, pero se pueden medir después de que el sistema esté en funcionamiento. Sin embargo, propiedades como la seguridad y la protección presentan diversos problemas. Aquí, se tiene conexión no sólo con un atributo relacionado con el comportamiento total del sistema, sino con el comportamiento que el sistema *no* debería mostrar. Un sistema seguro es aquel que no permite accesos no autorizados a sus datos, pero es claramente imposible predecir todos los posibles modos de acceso y prohibirlos de forma explícita. Por lo tanto, sólo es posible valorar estas propiedades por defecto. Esto es, sólo se puede saber que el sistema es inseguro cuando alguien lo viola.

2.2 Ingeniería de sistemas

La ingeniería de sistemas es la actividad de especificar, diseñar, implementar, validar, utilizar y mantener los sistemas socio-técnicos. Los ingenieros de sistemas no sólo tratan con el software, sino también con el hardware y las interacciones del sistema con los usuarios y su entorno. Deben pensar en los servicios que el sistema proporciona, las restricciones sobre las que el sistema se debe construir y funcionar y las formas en las que el sistema es usado para cumplir con su propósito. Como se ha tratado, los ingenieros de software necesitan tener conocimientos de ingeniería de sistemas, porque los problemas de la ingeniería del software son a menudo el resultado de decisiones de la ingeniería de sistemas (Thayer, 1997; Thayer, 2002).

Las fases del proceso de la ingeniería de sistemas se muestran en la Figura 2.2. Este proceso tuvo una influencia significativa en el modelo en «cascada» del proceso del software que se estudia en el Capítulo 4.

Existen diferencias importantes entre el proceso de la ingeniería de sistemas y el proceso de desarrollo del software:

1. *Alcance limitado para rehacer el trabajo durante el desarrollo de sistemas.* Una vez que se han tomado decisiones en la ingeniería del sistema, como la posición de una estación base en un sistema de telefonía móvil, cuesta mucho trabajo cambiarlas. Raramente es posible rehacer el trabajo en el diseño del sistema para resolver estos problemas. Una razón por la que el software ha llegado a ser tan importante en los sistemas es que permite cambios que se hacen durante el desarrollo del sistema, como respuesta a nuevos requerimientos.

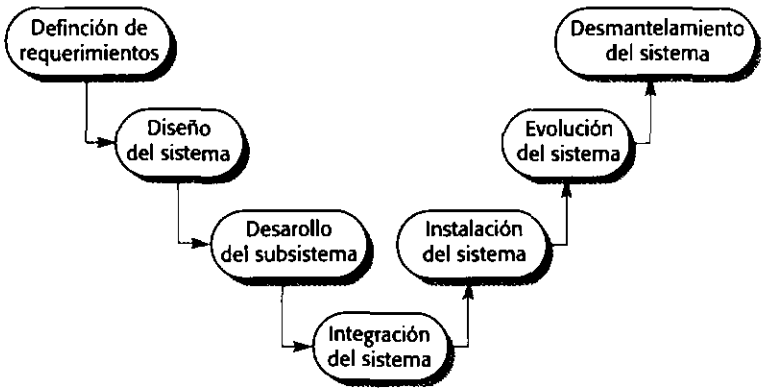


Figura 2.2 El proceso de la ingeniería de sistemas.

2. *Implicación interdisciplinaria.* Muchas disciplinas de la ingeniería se conjuntan en la ingeniería de sistemas. Existe una gran discrepancia debido a que diferentes ingenieros usan diferente terminología y convenciones.

La ingeniería de sistemas es una actividad interdisciplinaria que conjunta equipos de personas con diferentes bases de conocimiento. Los equipos de ingeniería de sistemas son necesarios debido al amplio conocimiento requerido para considerar todas las implicaciones de las decisiones en el diseño del sistema. Como ejemplo de esto, la Figura 2.3 muestra algunas de las disciplinas que conforman el equipo de ingeniería de sistemas para un sistema de control del tráfico aéreo (CTA) que utilizan radares y otros sensores para determinar la posición de los aviones.

Para muchos sistemas existen posibilidades casi infinitas de equilibrio entre los diferentes tipos de subsistemas. Las diferentes disciplinas negocian para decidir qué funcionalidad debe proporcionarse. A menudo no existe una decisión «correcta» sobre cómo se debe descomponer un sistema. Más bien, puede tener varias opciones, pero es posible que no pueda elegir la mejor solución técnica. Por ejemplo, una alternativa en un sistema de control del tráfico aéreo es construir nuevos radares, más que arreglar las instalaciones existentes. Si los ingenieros civiles relacionados con este proceso no tienen más trabajo que hacer, estarán a favor de esta opción debido a que les permite conservar sus trabajos. Pueden justificar esta elección utilizando argumentos técnicos.

2.2.1 Definición de requerimientos del sistema

Las definiciones de requerimientos del sistema especifican qué es lo que el sistema debe hacer (sus funciones) y sus propiedades esenciales y deseables. Como en el análisis de requerimien-

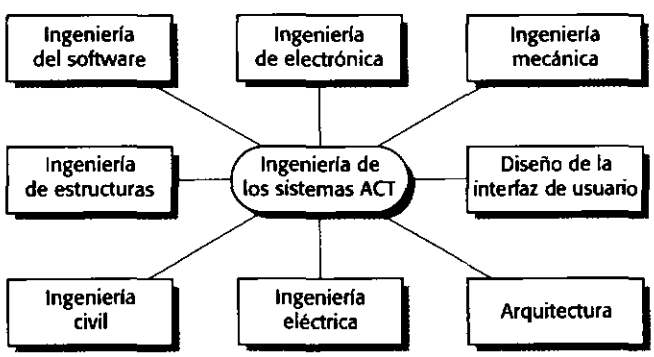


Figura 2.3. Disciplinas involucradas en la ingeniería de sistemas.

tos del software (tratado en la Parte 2), crear definiciones de requerimientos del sistema requiere consultar con los clientes del sistema y con los usuarios finales. Esta fase de definición de requerimientos usualmente se concentra en la derivación de tres tipos de requerimientos:

1. *Requerimientos funcionales abstractos.* Las funciones básicas que el sistema debe proporcionar se definen en un nivel abstracto. Una especificación más detallada de requerimientos funcionales tiene lugar en el nivel de subsistemas. Por ejemplo, en un sistema de control del tráfico aéreo, un requerimiento funcional abstracto especificaría que una base de datos del plan de vuelo debe usarse para almacenar los planes de vuelo de todos los aviones que entran al espacio aéreo controlado. Sin embargo, normalmente no se especificarían los detalles de la base de datos a menos que afecten a los requerimientos de otros subsistemas.
2. *Propiedades del sistema.* Como se señaló anteriormente, éstas son propiedades emergentes no funcionales del sistema, tales como la disponibilidad, el rendimiento y la seguridad. Estas propiedades no funcionales del sistema afectan a los requerimientos de todos los subsistemas.
3. *Características que no debe mostrar el sistema.* Algunas veces es tan importante especificar lo que el sistema no debe hacer como especificar lo que debe hacer. Por ejemplo, si está especificando un sistema de control del tráfico aéreo, puede especificar que el sistema no debe presentar demasiada información al controlador.

Una parte importante de la fase de definición de requerimientos es establecer un conjunto completo de objetivos que el sistema debe cumplir. Éstos no necesariamente deben expresarse forzosamente en términos de la funcionalidad del sistema, pero deben definir por qué se construye el sistema para un entorno particular.

Para ilustrar qué quiere decir esto, digamos que está especificando un sistema contra incendios y detección de intrusos para un edificio de oficinas. Un enunciado de los objetivos basado en la funcionalidad del sistema sería:

> *Construir un sistema de alarma contra incendios e intrusos para el edificio que proporcione avisos de fuego y de intrusiones no autorizadas tanto internas como externas.*

Este objetivo establece explícitamente que debe ser un sistema de alarma que proporcione avisos sobre eventos no deseados. Tal enunciado sería apropiado si estuviéramos reemplazando un sistema de alarma existente. En contraste, un enunciado más amplio de los objetivos sería:

> *Asegurar que el funcionamiento normal de los trabajos realizados en el edificio no se interrumpa por eventos como el fuego e intrusión no autorizada.*

Si enuncia el objetivo de esta forma, amplía y limita algunas decisiones en el diseño. Por ejemplo, este objetivo permite la protección contra intrusos utilizando tecnología sofisticada, sin alarma interna alguna. También puede excluir la utilización de extintores para la protección del fuego porque puede afectar a los sistemas eléctricos en el edificio e interrumpir seriamente el trabajo.

Una dificultad fundamental al establecer los requerimientos del sistema es que los problemas para los cuales se construyen los sistemas complejos son normalmente «problemas traviesos» (Rittel y Webber, 1973). Un «problema travieso» es un problema que es tan complejo, y en el que hay tantas entidades relacionadas, que no existe una especificación definitiva del problema. La verdadera naturaleza de éste emerge sólo cuando se desarrolla una solución. Un ejemplo extremo de un «problema travieso» es la previsión de terremotos. Nadie puede

predecir de forma precisa dónde será el epicentro de un terremoto, en qué momento ocurrirá o qué efecto tendrá en el entorno local. Por lo tanto, no es posible especificar completamente cómo abordar un terremoto. El problema sólo se puede abordar una vez que ha pasado.

2.2.2 Diseño del sistema

El diseño del sistema (Figura 2.4) se centra en proporcionar la funcionalidad del sistema a través de sus diferentes componentes. Las actividades que se realizan en este proceso son:

1. *Dividir requerimientos.* Analice los requerimientos y organícelos en grupos afines. Normalmente existen varias opciones posibles de división, y puede sugerir varias alternativas en esta etapa del proceso.
2. *Identificar subsistemas.* Debe identificar los diferentes subsistemas que pueden, individual o colectivamente, cumplir los requerimientos. Los grupos de requerimientos están normalmente relacionados con los subsistemas, de tal forma que esta actividad y la de división de requerimientos se pueden fusionar. Sin embargo, la identificación de subsistemas se puede ver influenciada por otros factores organizacionales y del entorno.
3. *Asignar requerimientos a los subsistemas.* Asigne los requerimientos a los subsistemas. En principio, esto debe ser sencillo si la división de requerimientos se utiliza para la identificación de subsistemas. En la práctica, no existe igualdad entre las divisiones de requerimientos y la identificación de subsistemas. Las limitaciones de los subsistemas comerciales pueden significar que tenga que cambiar los requerimientos para acomodarlos a estas restricciones.
4. *Especificar la funcionalidad de los subsistemas.* Debe enumerar las funciones específicas asignadas a cada subsistema. Esto puede verse como parte de la fase de diseño del sistema o, si el subsistema es un sistema de software, como parte de la actividad de especificación de requerimientos para ese sistema. También debe intentar especificar las relaciones entre los subsistemas en esta etapa.
5. *Definir las interfaces del subsistema.* Defina las interfaces necesarias y requeridas por cada subsistema. Una vez que estas interfaces se han acordado, es posible desarrollar estos subsistemas en paralelo.

Como se indica en las flechas bidireccionales en la Figura 2.4, en este proceso de diseño existe mucha realimentación e iteración de una etapa a la otra. Cuando surgen problemas y preguntas, a menudo tiene que rehacer el trabajo hecho en etapas anteriores.

Aunque se han separado los procesos de ingeniería de requerimientos y de diseño en este análisis, en la práctica están inextricablemente relacionados. Restricciones planteadas por sistemas existentes pueden limitar elecciones de diseño, y estas elecciones pueden ser especifi-

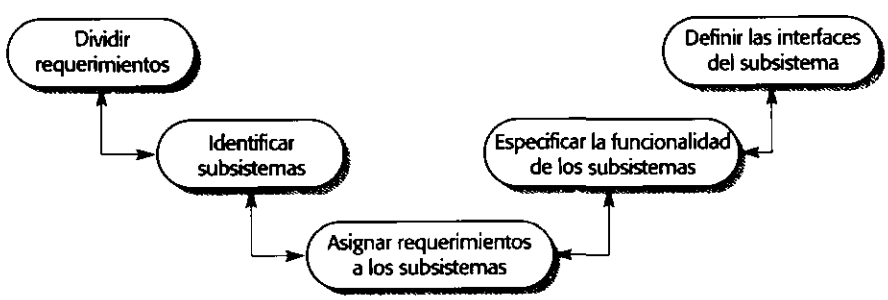


Figura 2.4
El proceso de diseño de sistemas.

cadas en los requerimientos. Puede tener que hacer algún diseño inicial para estructurar y organizar el proceso de la ingeniería de requerimientos. A medida que el proceso de diseño continúa, puede descubrir problemas con los requerimientos existentes y pueden surgir nuevos requerimientos. Por consiguiente, una manera de representar estos procesos relacionados es en forma de espiral, como se muestra en la Figura 2.5.

El proceso en espiral refleja la realidad de que los requerimientos afectan a las decisiones de diseño y viceversa, y de esta forma tiene sentido entrelazar estos procesos. Comenzando en el centro, cada vuelta de la espiral añade algún detalle a los requerimientos y al diseño. Algunas vueltas se centran en los requerimientos; otras, en el diseño. A veces, nuevo conocimiento recopilado durante los procesos de requerimientos y diseño significa que la declaración del problema en sí misma tiene que ser cambiada.

Para la mayoría de los sistemas, existen muchos diseños posibles que cumplen los requerimientos. Éstos comprenden una amplia gama de soluciones que combinan hardware, software y operaciones humanas. La solución que elija para el desarrollo futuro deberá ser la solución técnica más apropiada que cumpla los requerimientos. Sin embargo, las intervenciones organizacionales y políticas pueden influir en la elección de la solución. Por ejemplo, un cliente del gobierno puede preferir usar proveedores nacionales antes que los extranjeros para su sistema, aun cuando el producto nacional sea técnicamente inferior. Estas influencias normalmente tienen efecto en la fase de revisión y valoración en el modelo en espiral, donde los diseños y requerimientos pueden ser aceptados o rechazados. El proceso finaliza cuando la revisión y evaluación muestra que los requerimientos y el diseño de alto nivel son suficientemente detallados para permitir empezar la siguiente fase del proceso.

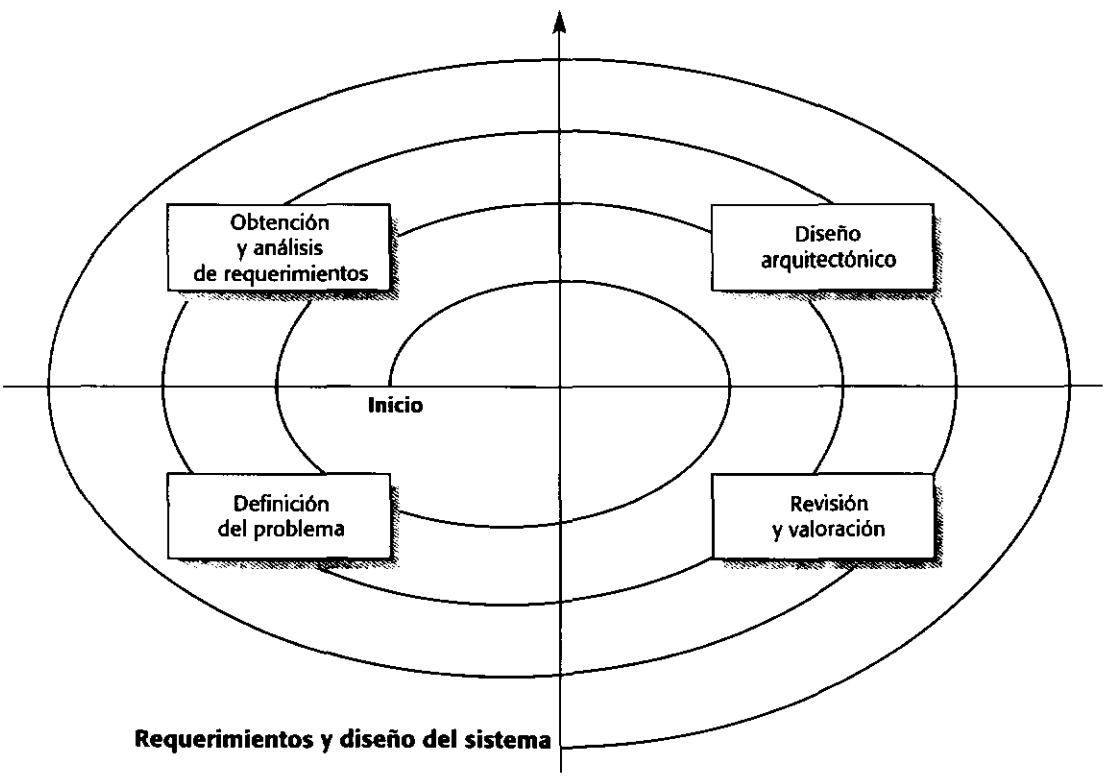


Figura 2.5 Un modelo en espiral de requerimientos y diseño.

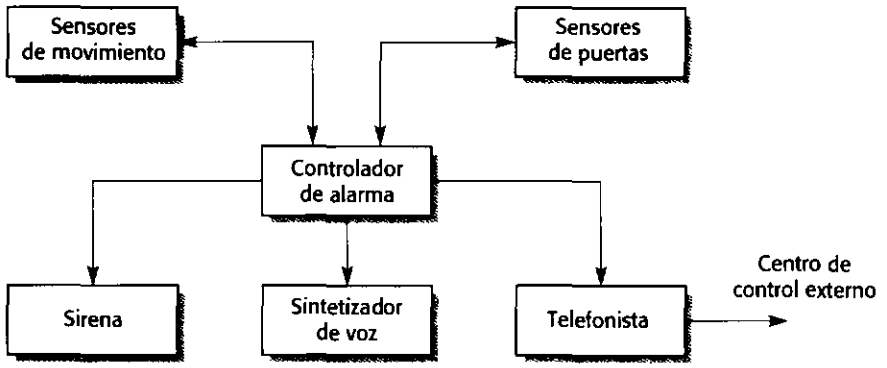


Figura 2.6
Un sistema sencillo de alarma contra ladrones.

2.2.3 Modelado de sistemas

Durante la actividad de requerimientos y diseño del sistema, éstos pueden ser modelados como un conjunto de componentes y de relaciones entre estos componentes. Esto se puede ilustrar gráficamente en un modelo arquitectónico del sistema, el cual proporciona al lector una visión general de la organización del sistema.

La arquitectura del sistema puede se presentada como un diagrama de bloques que muestra los principales subsistemas y la interconexión entre ellos. Al dibujar un diagrama de bloque, debe representar cada subsistema mediante un rectángulo, y debe mostrar las relaciones entre los subsistemas usando flechas que unan estos rectángulos. Las relaciones pueden ser flujo de datos, «utiliza»/«utilizado por» o algún otro tipo de relación dependiente.

Por ejemplo, en la Figura 2.6 se muestra la descomposición de un sistema de alarma contra intrusos en sus componentes principales. El diagrama de bloques debe complementarse con una breve descripción de cada subsistema, como se muestra en la Figura 2.7.

En este nivel de detalle, el sistema se descompone en un conjunto de subsistemas que interactúan. Cada uno de éstos debe ser representado de forma similar hasta que el sistema esté dividido en componentes funcionales. Éstos son componentes que, cuando se ven desde la perspectiva del subsistema, proporcionan una función única. En contraste, un sistema comúnmente es multifuncional. Por supuesto, cuando se ve desde otra perspectiva (por ejemplo, desde los componentes del fabricante), un componente funcional es un sistema.

Históricamente, el modelo de arquitectura de sistemas fue utilizado para identificar componentes de hardware y software que podían desarrollarse en paralelo. Sin embargo, esta distinción hardware/software se ha hecho irrelevante. En la actualidad, la mayoría de los componentes incluyen algunas capacidades de cómputo. Por ejemplo, las máquinas para enlazar

Sensores de movimiento	**Detecta el movimiento en los cuartos vigilados por el sistema.**
Sensores de puertas	**Detecta la apertura de puertas externas del edificio.**
Controlador de alarma	**Controla la operación del sistema.**
Sirena	**Emite avisos auditivos cuando existen intrusos.**
Sintetizador de voz	**Sintetiza un mensaje de voz proporcionando la ubicación del intruso.**
Telefonista	**Hace llamadas externas para notificar a seguridad, a la policía, etc.**

Figura 2.7 Descripción de los subsistemas en el sistema de alarma contra ladrones.

redes consisten en cables físicos, repetidores y gateways. Éstos incluyen procesadores y software para manejar estos procesadores, así como componentes electrónicos especializados.

En el nivel de la arquitectura, es más apropiado clasificar los subsistemas de acuerdo con su función antes de tomar decisiones sobre el tipo de hardware/software. La decisión de proporcionar una función mediante hardware o software depende de factores no técnicos, como la disponibilidad de componentes comerciales o el tiempo disponible para desarrollar el componente.

Los diagramas de bloques se pueden utilizar para sistemas de cualquier tamaño. La Figura 2.8 muestra la arquitectura de un sistema de mayor tamaño para el control del tráfico aéreo. Varios subsistemas principales mostrados son a su vez sistemas grandes. Las flechas que conectan estos sistemas muestran el flujo de información entre estos subsistemas.

2.2.4 Desarrollo de los subsistemas

Durante el desarrollo de los subsistemas, se implementan los que se hayan identificado durante el diseño del sistema. Esto implica comenzar otro proceso de la ingeniería de sistemas para los subsistemas individuales o, si el subsistema es software, un proceso de software que comprende requerimientos, diseño, implementación y pruebas.

Ocasionalmente, todos los subsistemas son desarrollados desde sus inicios durante el proceso de desarrollo. Sin embargo, normalmente algunos de estos subsistemas son comerciales (COTS), los cuales se compran para integrarse en el sistema. Normalmente es mucho más barato comprar productos existentes que desarrollar componentes de propósito especial. En esta

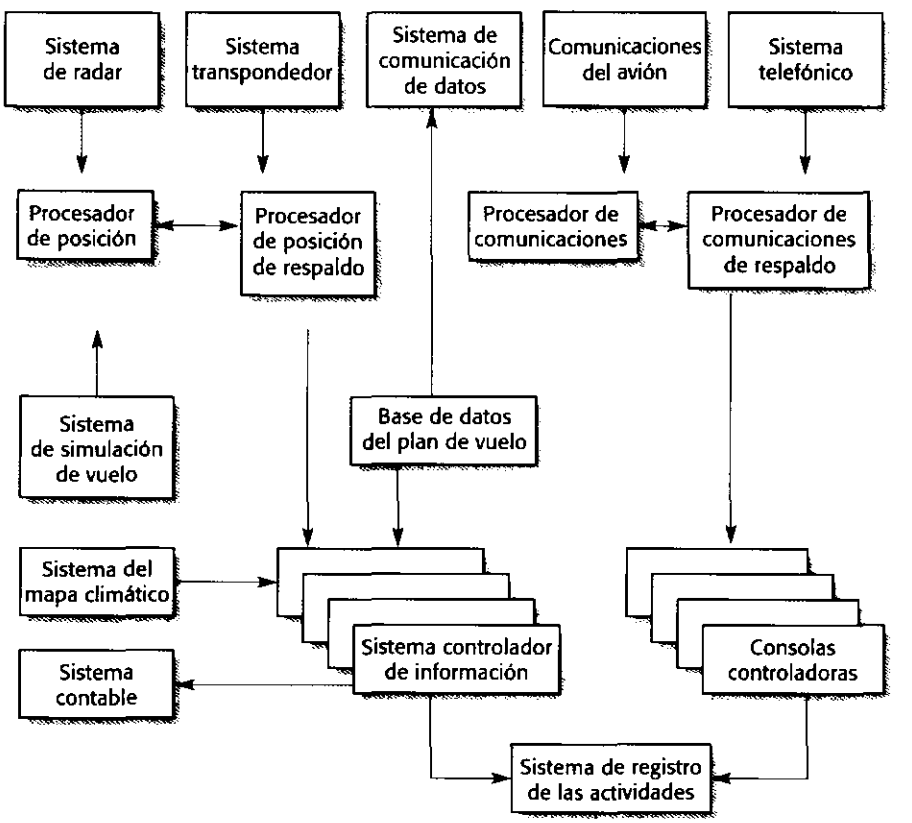


Figura 2.8 Un modelo arquitectónico para un sistema de control del tráfico aéreo.

etapa, puede tener que entrar de nuevo en la actividad de diseño para acomodar un componente comprado. Los sistemas COTS pueden no cumplir exactamente los requerimientos, pero, si los productos comerciales están disponibles, es mucho mejor volver a pensar el diseño.

Es común que los subsistemas se desarrollen en paralelo. Cuando se encuentran problemas que sobrepasan los límites del subsistema, se debe realizar una petición de modificación del sistema. Si los sistemas requieren una amplia ingeniería del hardware, puede resultar muy caro hacer modificaciones después de haberse iniciado su fabricación. A menudo se deben realizar «revisiones de trabajo» con el fin de detectar los problemas. Estas «revisiones de trabajo» comúnmente implican cambios en el software debido a la flexibilidad inherente a él. Esto conduce a cambios en los requerimientos del software; por lo tanto, como se explicó en el Capítulo 1, es importante diseñar software para el cambio, de modo que puedan implementarse nuevos requerimientos sin un excesivo coste adicional.

2.2.5 Integración del sistema

Durante el proceso de integración del sistema, se toman los subsistemas desarrollados de forma independiente y se conjuntan para crear el sistema completo. La integración se puede hacer utilizando el enfoque del «big bang», que consiste en integrar todos los subsistemas al mismo tiempo. Sin embargo, a efectos técnicos y de administración, el mejor enfoque es un proceso de integración creciente donde los sistemas se integran uno a uno, por dos razones:

1. Por lo general, es imposible confeccionar una agenda para el desarrollo de todos los subsistemas de tal forma que todos terminen al mismo tiempo.
2. La integración creciente reduce el costo en la localización de errores. Si varios subsistemas se integran simultáneamente, un error que surja durante las pruebas puede estar en cualquiera de estos subsistemas. Cuado un único subsistema se integra en un sistema en funcionamiento, los errores que se produzcan estarán probablemente en el subsistema recién integrado o en las interacciones entre los subsistemas existentes y el nuevo sistema.

Una vez que los componentes han sido integrados, tiene lugar un extenso programa de pruebas del sistema. Estas pruebas pretenden probar las interfaces entre los componentes y el comportamiento del sistema en su totalidad.

Los defectos de los subsistemas que son consecuencia de suposiciones inválidas en los otros subsistemas, a menudo aparecen durante la integración del sistema. Esto puede conducir a problemas entre los diferentes contratistas responsables de los diferentes subsistemas. Cuando se descubren problemas en la interacción de subsistemas, los diferentes contratistas pueden discutir sobre qué subsistema es el defectuoso. Las negociaciones de cómo solucionar el problema pueden llevar semanas o meses.

Como cada vez más los sistemas son construidos por la integración de componentes hardware y software COTS, la integración de sistemas está adquiriendo una importancia creciente. En algunos casos, no hay separación en el desarrollo de subsistemas y la integración es, esencialmente, la fase de implementación del sistema.

2.2.6 Evolución del sistema

Los sistemas grandes y complejos tienen un periodo de vida largo. Durante su vida, se cambian para corregir errores en los requerimientos del sistema original y para implementar nuevos requerimientos que surgen. Los sistemas de cómputo se reemplazan por nuevas máqui-

nas más rápidas. La organización que utiliza el sistema puede reorganizarse y utilizar el sistema de forma diferente. El entorno externo del sistema puede cambiar, forzando cambios en el sistema.

La evolución del sistema, como la del software (expuesta en el Capítulo 21), es inherentemente costosa, por varias razones:

1. Los cambios propuestos tienen que analizarse cuidadosamente desde perspectivas técnicas y de negocios. Los cambios tienen que contribuir a los objetivos del sistema y no deben tener simplemente una motivación técnica.
2. Debido a que los subsistemas nunca son completamente independientes, los cambios en uno pueden afectar de forma adversa al funcionamiento o comportamiento de otros. Por lo tanto, será necesario cambiar estos subsistemas.
3. A menudo no se registran las razones del diseño original. Los responsables de la evolución del sistema tienen que resolver por qué se tomaron decisiones particulares de diseño.
4. Al paso del tiempo, su estructura se corrompe por el cambio de tal forma que se incrementan los costos de cambios adicionales.

Los sistemas que se han desarrollado con el tiempo dependen de tecnologías hardware y software obsoletas. Si tienen un papel crítico en la organización, son conocidos como sistemas heredados —sistemas que a la organización le gustaría reemplazar pero donde los riesgos de introducir un nuevo sistema son altos—. En la Sección 2.4 se tratan algunas cuestiones de los sistemas heredados.

2.2.7 Desmantelamiento del sistema

El desmantelamiento del sistema significa poner fuera de servicio a dicho sistema después de que termina su periodo de utilidad operativa. Para sistemas hardware esto puede implicar el desmontaje y reciclaje de materiales o el tratamiento de sustancias tóxicas. El software no tiene problemas físicos de desmantelamiento, pero algún software se puede incorporar en un sistema para ayudar al proceso de desmantelamiento. Por ejemplo, el software se puede utilizar para controlar el estado de componentes hardware. Cuando el sistema se desmantela, los componentes en buen estado se pueden identificar y reutilizar en otros sistemas.

Si los datos del sistema que se está desmantelando todavía poseen valor para su organización, puede tener que convertirlos para utilizarlos en otros sistemas. A menudo esto implica un costo, ya que la estructura de datos puede estar implícitamente definida en el software mismo. Debe analizar el software para descubrir cómo están estructurados los datos y entonces escribir un programa para reorganizarlos en las estructuras exigidas por el nuevo sistema.

2.3 Organizaciones, personas y sistemas informáticos

Los sistemas socio-técnicos son sistemas empresariales que tiene la intención de ayudar a conseguir algunos objetivos organizacionales o de negocio. Esto puede ser incrementar las ventas, reducir el uso de material en la fabricación, recaudar impuestos, mantener un espacio aéreo seguro, etc. Puesto que están dentro de un entorno organizacional, la consecución, desarrollo y uso de estos sistemas están influenciados por las políticas y procedimientos de la

organización y por su cultura de trabajo. Los usuarios del sistema son personas que están influenciadas por la forma en la que es gestionada la organización y por sus relaciones con otras personas dentro y fuera ésta.

Por lo tanto, cuando está intentando entender los requerimientos para un sistema socio-técnico necesita entender su entorno organizacional. De lo contrario, los sistemas pueden no cumplir las necesidades del negocio, y los usuarios y sus directivos puede rechazar el sistema.

Los factores humanos y organizacionales del entorno del sistema que afectan a su diseño son los siguientes:

1. *Cambios en el proceso.* ¿El sistema requiere cambiar los procesos en el entorno? Si es así, se necesitará formación. Si los cambios son significativos, o implican que la gente pierda su trabajo, existe el peligro de que los usuarios se resistan a la introducción del sistema.
2. *Cambios en el trabajo.* ¿El sistema inhabilita a los usuarios en un entorno o hace que cambie su forma de trabajar? Si es así, se resistirán a la introducción del sistema en la organización. Los diseños que implican un cambio en las formas de trabajo de los directivos con el fin de adaptarse al sistema informático tienen sus implicaciones. Los directivos sienten que su jerarquía en la organización se ve reducida por el sistema.
3. *Cambios organizacionales.* ¿El sistema cambia la estructura de poder en una organización? Por ejemplo, si una organización depende de un sistema complejo, aquellos que saben cómo operar el sistema tienen un gran poder político.

Estos factores humanos, sociales y organizacionales son a menudo críticos para determinar si un sistema cumple con éxito sus objetivos. Desgraciadamente, predecir sus efectos sobre los sistemas es una tarea difícil para los ingenieros que tienen poca experiencia en estudios sociales o culturales. Para ayudar a entender los efectos de los sistemas en las organizaciones, se han desarrollado varias metodologías, como la Sociotécnica de Mumford (Mumford, 1989) y la Metodología de Sistemas Suaves de Checkland (Checkland y Scholes, 1990; Checkland, 1981). También existen amplios estudios sociológicos de los efectos en el trabajo de los sistemas informáticos (Ackroyd *et al.*, 1992).

De forma ideal, todo el conocimiento organizacional relevante debe incluirse en la especificación del sistema, de tal forma que los diseñadores puedan tenerlo en cuenta. En realidad, esto es imposible. Los diseñadores de sistemas tienen que hacer suposiciones basadas en otros sistemas comparables y en el sentido común. Si sus suposiciones son erróneas, el sistema puede funcionar mal de forma impredecible. Por ejemplo, si los diseñadores de un sistema no comprenden que las diferentes partes de una organización realmente pueden tener objetivos contradictorios, entonces cualquier sistema que sea desarrollado para una organización inevitablemente tendrá algún usuario insatisfecho.

2.3.1 Procesos organizacionales

En la Sección 2.2, se introdujo el modelo de procesos de la ingeniería de sistemas que mostró los subprocesos implicados en el desarrollo del sistema. Sin embargo, el proceso de desarrollo no es el único proceso implicado en la ingeniería de sistemas. Éste se relaciona con el proceso de adquisición del sistema y con el proceso de uso y operación del sistema, como se ilustra en la Figura 2.9.

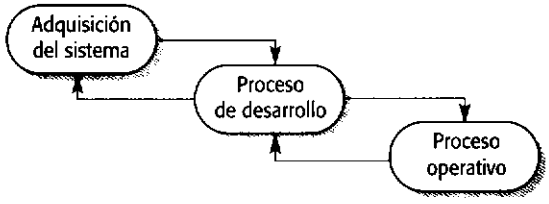


Figura 2.9 Procesos de adquisición, desarrollo y operativo.

El proceso de adquisición normalmente está contenido dentro de la organización que comprará y usará el sistema (la organización cliente). El proceso de adquisición del sistema está relacionado con la toma de las decisiones sobre la mejor forma en la que una organización puede adquirir un sistema y decidir sobre los mejores proveedores de ese sistema.

Los sistemas grandes y complejos comúnmente consisten en una mezcla de componentes comerciales y componentes construidos de forma especial. Una razón por la cual se incluye cada vez más software en los sistemas es que permite una mayor utilización de los diferentes componentes hardware existentes, donde el software actúa como un «pegamento» para hacer que estos componentes hardware trabajen juntos de forma efectiva. La necesidad de desarrollar este «pegamento» se debe a que el ahorro por la utilización de los componentes comerciales no es tan grande como se supone. En el Capítulo 18 se tratan con más detalle los sistemas COTS.

La Figura 2.10 muestra el proceso de adquisición tanto para sistemas existentes como para sistemas especialmente diseñados. Algunos puntos importantes del proceso mostrado en este diagrama son los siguientes:

1. Comúnmente los componentes comerciales no cumplen de forma exacta los requerimientos, a menos que éstos se hayan escrito teniendo en cuenta dichos componentes. Por lo tanto, elegir un sistema significa que tiene que encontrar la correspondencia más cercana entre los requerimientos del sistema y las funcionalidades ofrecidas por los sistemas comerciales. Puede tener entonces que modificar los requerimientos y esto puede producir efectos perturbadores en otros subsistemas.
2. Cuando un sistema se construye de forma especial, la especificación de requerimientos actúa como la base de un contrato para la adquisición del sistema. Es, por lo tanto, un documento legal, y también técnico.
3. Una vez que se ha seleccionado un contratista para construir el sistema, existe un periodo de negociación del contrato en el cual puede tener que negociar nuevos cambios en los requerimientos y discutir temas como el costo de los cambios del sistema.

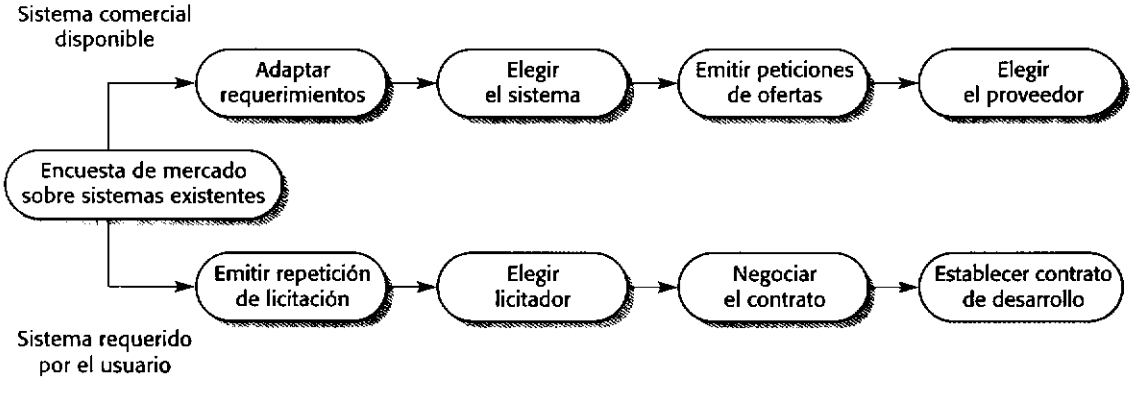


Figura 2.10 El proceso de adquisición del sistema.

Se han resumido las fases principales del proceso de desarrollo del sistema. Los sistemas complejos normalmente son desarrollados por una organización diferente (el proveedor) de la organización que adquiere el sistema. La razón de esto es que rara vez el negocio del solicitante es el desarrollo de sistemas, por lo que sus empleados no tienen la capacidad necesaria para desarrollar por sí mismos sistemas complejos. De hecho, muy pocas organizaciones tienen la capacidad de desarrollar, fabricar y probar todos los componentes de un sistema largo y complejo.

El proveedor, a quien normalmente se le conoce como el contratista principal, puede subcontratar el desarrollo de diferentes subsistemas a un cierto número de subcontratistas. Para sistemas grandes, como los de control del tráfico aéreo, un grupo de proveedores puede formar un consorcio para competir por el contrato. El consorcio debe incluir todas las capacidades requeridas por este tipo de sistema, como proveedores de hardware, desarrolladores de software y proveedores de periféricos y de equipo especial, como radares.

El solicitante trata con el contratista en vez de con los subcontratistas, por lo que hay una única interfaz solicitante/proveedor. Los subcontratistas diseñan y construyen partes del sistema de acuerdo con una especificación producida por el contratista principal. Una vez completadas, el contratista principal integra estos componentes y los entrega al cliente que compra el sistema. Dependiendo del contrato, el solicitante puede permitir al contratista elegir libremente a los subcontratistas o puede exigirle que los elija de una lista aprobada.

Los procesos operativos son los procesos que están relacionados con el uso del sistema para su propósito definido. Por ejemplo, los operadores de un sistema de control del tráfico aéreo siguen procesos específicos cuando el avión entra y sale del espacio aéreo, cuando tienen que cambiar la altura o la velocidad, cuando ocurre una emergencia, etc. Para sistemas nuevos, estos procesos operativos tienen que ser definidos y documentados durante el proceso de desarrollo del sistema. Puede que los operadores tengan que formarse y haya que adaptar otros procesos de trabajo para hacer efectivo el uso del nuevo sistema. En esta etapa pueden surgir problemas no detectados, porque la especificación del sistema puede contener errores u omisiones. Mientras que el sistema puede funcionar conforme a la especificación, sus funciones pueden no cumplir las necesidades operativas reales. Por consiguiente, es posible que los operadores no usen el sistema como sus diseñadores pensaron.

La ventaja clave de tener gente en un sistema es que ésta tiene una capacidad única para responder eficazmente a situaciones inesperadas, aun cuando no hayan tenido una experiencia directa en estas situaciones. Por lo tanto, cuando las cosas van mal, los operadores pueden a menudo recuperar la situación, aunque algunas veces esto pueda significar que no se cumplan los procesos definidos. Los operadores también usan su conocimiento local para adaptar y mejorar los procesos. Normalmente, el proceso operativo real es diferente del anticipado por los diseñadores del sistema.

Esto significa que los diseñadores deben diseñar los procesos operativos para ser flexibles y adaptables. Los procesos operativos no deben ser demasiado restrictivos, ni requerir operaciones hechas en un orden en particular, y el software del sistema no debe depender de que no se siga un proceso específico. Los operadores normalmente mejoran el proceso porque saben qué es lo que funciona y lo que no funciona en una situación real.

Una cuestión que puede surgir solamente después de que el sistema entre en funcionamiento es el problema del funcionamiento del nuevo sistema junto a sistemas existentes. Es posible que existan problemas físicos de incompatibilidad, o que sea difícil el transferir datos de un sistema al otro. Pueden surgir problemas más sutiles debido a que diferentes sistemas tienen distintas interfaces de usuario. Introducir el nuevo sistema puede incrementar el índice de error del operador para los sistemas existentes porque los operadores confunden los comandos de la interfaz de usuario.

2.4 Sistemas heredados

Debido al tiempo y esfuerzo requeridos para desarrollar un sistema complejo, los grandes sistemas informáticos normalmente tienen un tiempo de vida largo. Por ejemplo, los sistemas militares son diseñados normalmente para un tiempo de vida de 20 años, y muchos de los sistemas de control del tráfico aéreo del mundo todavía dependen de software y procesos operativos que fueron desarrollados en un principio en los años 60 y 70. Algunas veces es demasiado caro y peligroso descartar tales sistemas de negocio críticos después de unos pocos años de uso. Su desarrollo continúa durante toda su vida con cambios para satisfacer nuevos requerimientos, nuevas plataformas operativas, y así sucesivamente.

Los sistemas heredados son sistemas informáticos socio-técnicos que han sido desarrollados en el pasado, a menudo usando una tecnología antigua y obsoleta. Estos sistemas no solamente incluyen hardware y software sino también procesos y procedimientos heredados —antiguas formas de hacer cosas que son difíciles de cambiar porque dependen de software heredado—. Cambios en una parte del sistema inevitablemente implican cambios en otros componentes.

Los sistemas heredados son a menudo sistemas de negocio críticos. Se mantienen porque es demasiado arriesgado reemplazarlos. Por ejemplo, para la mayoría de los bancos el sistema contable de clientes fue uno de los primeros sistemas. Las políticas y procedimientos organizacionales pueden depender de este sistema. Si el banco fuera a descartar y reemplazar el sistema contable de clientes (el cual es posible que se ejecute en un costoso hardware mainframe), entonces habría un serio riesgo de negocio si el sistema de recambio no funcionara adecuadamente. Además, los procedimientos existentes tendrían que cambiar, y esto puede molestar a las personas de la organización y causar dificultades con los auditores del banco.

La Figura 2.11 ilustra las partes lógicas de un sistema heredado y sus relaciones:

1. *Sistema hardware*. En muchos casos, los sistemas heredados se crearon para hardware mainframe que ya no está disponible, es costoso de mantener y no es compatible con las actuales políticas de compras de IT organizacionales.
2. *Software de apoyo*. Los sistemas heredados cuentan con una gran variedad de software de apoyo que van desde sistemas operativos y utilidades suministradas por el fabricante de hardware hasta los compiladores utilizados para el desarrollo de sistemas. De nuevo, éstos pueden ser obsoletos o ya no recibir soporte de sus proveedores originales.
3. *Software de aplicación*. El sistema de aplicación que proporciona los servicios del negocio por lo general está compuesto de varios programas independientes desarrolla-

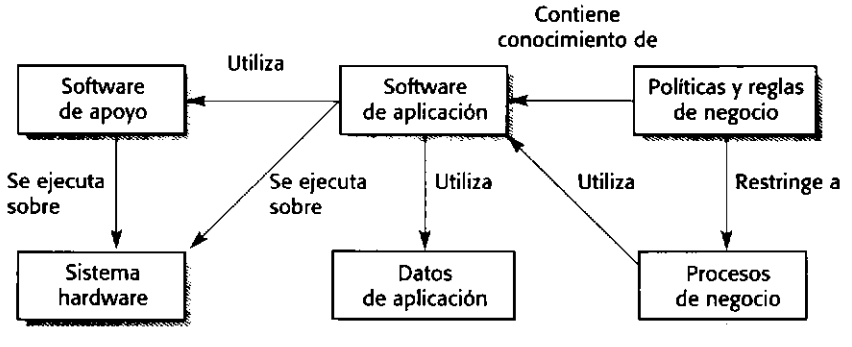


Figura 2.11
Componentes de los sistemas heredados.

dos en momentos diferentes. Algunas veces, el término sistema heredado significa este software de aplicación en lugar del sistema completo.

4. *Datos de aplicación.* Son los datos procesados por el sistema de aplicación. En muchos sistemas heredados, se ha acumulado un inmenso volumen de datos a lo largo del tiempo de vida del sistema. Estos datos pueden ser incongruentes y estar duplicados en varios archivos.
5. *Procesos de negocio.* Son los procesos utilizados en los negocios para lograr algún objetivo del negocio. Un ejemplo de un proceso de negocio en una compañía de seguros sería emitir una política de seguros; en una fábrica, un proceso de negocio sería aceptar un pedido para los productos y estipular el proceso de fabricación asociado. Los procesos de negocio pueden ser diseñados alrededor de un sistema heredado y restringidos por la funcionalidad que éste proporciona.
6. *Políticas y reglas de negocio.* Son las definiciones de cómo llevar a cabo los negocios y las restricciones sobre éstos. La utilización del sistema de aplicación heredado está contenida en estas políticas y reglas.

Una forma alternativa de observar estos componentes de un sistema heredado es como una serie de capas, según se muestra en la Figura 2.12. Cada capa depende de la capa inmediatamente inferior y tiene una interfaz con esa capa. Si las interfaces se mantienen, entonces podrían hacerse cambios dentro de una capa sin afectar a las adyacentes.

En la práctica, esta visión simple rara vez funciona, y los cambios en una capa del sistema requieren cambios consecuentes en las capas inferiores y superiores al nivel que se cambian. Las razones de esto son las siguientes:

1. Cambiar una capa en el sistema puede introducir nuevos recursos, y las capas más altas en el sistema se pueden entonces cambiar para aprovechar estos recursos. Por ejemplo, introducir una nueva base de datos en la capa del software de apoyo puede incluir recursos para acceder a los datos a través de una navegador web, y los procesos de negocio pueden ser modificados para aprovechar este recurso.
2. Cambiar el software puede ralentizar el sistema, por lo que se necesita un nuevo hardware para mejorar el rendimiento del sistema. El incremento en el rendimiento a partir del nuevo hardware puede significar que los cambios de software adicionales que anteriormente no eran prácticos ahora son posibles.
3. A menudo, es imposible dar mantenimiento a las interfaces de hardware, especialmente si se propone un cambio radical a un nuevo tipo de hardware. Por ejemplo, si una compañía pasa de un hardware mainframe a sistemas cliente-servidor (discutidos en el Capítulo 11), éstos por lo general tienen diferentes sistemas operativos. Por lo tanto, se pueden requerir cambios mayores en el software de aplicación.

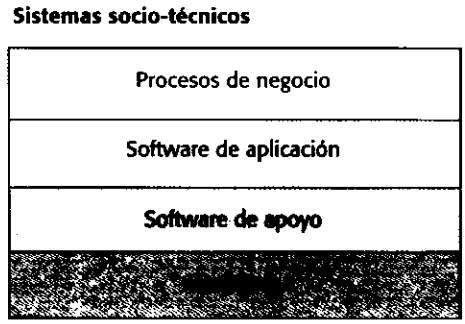


Figura 2.12 Modelo de capas de un sistema heredado.

PUNTOS CLAVE

- Los sistemas socio-técnicos incluyen hardware, software y personas, y se sitúan dentro de una organización. Están diseñados para ayudar a la organización a cumplir algún objetivo amplio.
- Las propiedades emergentes de un sistema son características de los sistemas como un todo más que sus partes componentes. Incluyen propiedades como el rendimiento, la fiabilidad, la usabilidad, la seguridad y la protección. El éxito o fracaso de un sistema depende a menudo de estas propiedades emergentes.
- El proceso de la ingeniería de sistemas comprende la especificación, el diseño, el desarrollo, la integración y las pruebas. La integración de sistemas es crítica cuando diversos subsistemas de diferentes proveedores deben trabajar de manera conjunta.
- Factores humanos y organizacionales como la estructura y políticas organizacionales influyen de forma significativa en el funcionamiento de los sistemas socio-técnicos.
- Dentro de una organización, existen complejas relaciones entre los procesos de adquisición, desarrollo y operativo del sistema.
- Un sistema heredado es un sistema antiguo que aún proporciona servicios esenciales de negocio.
- Los sistemas heredados no son sólo sistemas de software de aplicación. Son sistemas socio-técnicos, por lo que incluyen procesos de negocio, software de aplicación, software de apoyo y sistema hardware.

LECTURAS ADICIONALES

«Software system engineering: A tutorial». Una buena visión general de la ingeniería de sistemas, aunque Thayer se centra exclusivamente en los sistemas informáticos y no trata asuntos socio-técnicos. (R. H. Thayer, *IEEE Computer*, abril 2002.)

«Legacy information systems: Issues and directions». Una visión general de los problemas de los sistemas heredados con atención particular a los problemas de datos heredados. (J. Bisbal *et al.*, *IEEE Software*, septiembre-octubre 1999.)

Systems Engineering: Coping with Complexity. En el momento de escribir la presente obra, éste era aún el mejor libro de ingeniería de sistemas disponible. Se centra en los procesos de la ingeniería de sistemas con buenos capítulos sobre requerimientos, arquitectura y gestión de proyectos. (R. Stevens *et al.*, 1998, Prentice-Hall.)

«Airport 95: Automated baggage system». Un caso de estudio excelente y legible de lo que puede resultar mal en un proyecto de ingeniería de sistemas y cómo el software tiende a tener la culpa de los grandes fallos de los sistemas. (*ACM Software Engineering Notes*, 21 de marzo de 1996.)

EJERCICIOS

2.1 Explique qué otros sistemas dentro del entorno del sistema pueden tener efectos no previstos en su funcionamiento.

2.2 Explique por qué especificar un sistema para ser utilizado por los servicios de emergencia en la gestión de desastres es un problema travieso.

2.3 Mencione la manera en que los sistemas de software utilizados en un automóvil pueden ayudar al desmantelamiento (desechos) del sistema completo.

2.4 Explique por qué es importante presentar una descripción completa de una arquitectura del sistema en una etapa inicial del proceso de especificación del sistema.

2.5 Considere un sistema de seguridad que es una versión extendida del sistema mostrado en la Figura 2.6, que está pensado para proteger contra la intrusión y para detectar fuego. Contiene sensores de humo, de movimiento y de puertas, videocámaras controladas por computadora, que se encuentran en varios lugares del edificio, una consola de operación donde se informa del estado del sistema, y facilidades de comunicación externa para llamar a los servicios apropiados como la policía y los bomberos. Dibuje un diagrama de bloques de un posible diseño de dicho sistema.

2.6 Se construye un sistema de detección de inundaciones para avisar de posibles inundaciones en lugares que se ven amenazados por éstas. El sistema incluirá un conjunto de sensores para vigilar el cambio en los niveles del río, vínculos a un sistema meteorológico que proporciona la previsión del tiempo, vínculos a los sistemas de comunicación de los servicios de emergencia (policía, guardacostas, etc.), monitores de vídeo instalados en lugares específicos, un cuarto de control equipado con consolas de operación y monitores de vídeo.

Los controladores pueden acceder a la información de la base de datos y emitir pantallas de vídeo. El sistema de base de datos incluye información de los sensores, la ubicación de los sitios en riesgo y las condiciones de amenaza para estos sitios (por ejemplo, marea alta, vientos del suroeste, etc.), tablas de las mareas para los sitios costeros, el inventario y localización del equipo de control de inundaciones, detalle de los contactos de los servicios de emergencia, estaciones locales de radio, etc.

Dibuje un diagrama de bloques de una posible arquitectura para dicho sistema. Debe identificar los subsistemas principales y los vínculos entre ellos.

2.7 Un consorcio de museos europeos va a desarrollar un sistema multimedia de museo virtual que ofrece experiencias virtuales de la Grecia antigua. El sistema debe proporcionar a los usuarios la función de ver modelos 3-D de la Grecia antigua a través de un navegador web estándar y también debe apoyar una experiencia de realidad virtual. ¿Qué dificultades políticas y organizacionales pueden surgir cuando el sistema se instale en los museos que forman el consorcio?

2.8 Explique por qué los sistemas heredados pueden ser críticos en el funcionamiento de un negocio.

2.9 Explique por qué los sistemas heredados pueden causar dificultades para las compañías que desean reorganizar sus procesos de negocio.

2.10 ¿Cuáles son los argumentos a favor y en contra para considerar que la ingeniería de sistemas es una profesión, como la ingeniería eléctrica o la de software?

2.11 Suponga que es un ingeniero relacionado con el desarrollo de un sistema financiero. Durante la instalación, descubre que el sistema hará que se prescindan de muchas personas. La gente del entorno le niega el acceso a información esencial para completar la instalación del sistema. ¿Hasta dónde debería, como ingeniero de sistemas, verse envuelto en esto? ¿Es responsabilidad suya completar la instalación como lo estipula el contrato? ¿Debería abandonar el trabajo hasta que la organización haya resuelto el problema?

3 Sistemas críticos

Objetivos

El objetivo de este capítulo es introducir el concepto de sistema crítico, cuya característica más importante es la confiabilidad. Cuando haya leído este capítulo:

- comprenderá que en un sistema crítico un fallo de funcionamiento del sistema puede tener graves consecuencias humanas o económicas;
- comprenderá las cuatro dimensiones de la confiabilidad de un sistema: disponibilidad, fiabilidad, seguridad y protección;
- comprenderá que para lograr la confiabilidad se tienen que evitar los errores durante el desarrollo de un sistema, detectar y eliminar errores cuando el sistema se está utilizando y limitar el daño ocasionado por fallos operacionales.

Contenidos

Los fallos de funcionamiento del software son relativamente comunes. En la mayoría de los casos, estos fallos provocan molestias, pero no daños graves ni a largo plazo. Sin embargo, en algunos sistemas un fallo de funcionamiento puede ocasionar pérdidas económicas significativas, daño físico o amenazas a la vida humana. Estos sistemas se conocen como sistemas críticos. Los sistemas críticos son sistemas técnicos o socio-técnicos de los cuales dependen las personas o los negocios. Si estos sistemas no ofrecen sus servicios de la forma esperada, pueden provocar graves problemas y pérdidas importantes.

Hay tres tipos principales de sistemas críticos:

1. *Sistemas de seguridad críticos.* Son sistemas cuyo fallo de funcionamiento puede provocar perjuicio, pérdida de vidas o daños graves al medio ambiente. Un ejemplo de un sistema de seguridad crítico es un sistema de control para una planta de fabricación de productos químicos.
2. *Sistemas de misión críticos.* Son sistemas cuyo fallo de funcionamiento puede provocar errores en algunas actividades dirigidas por objetivos. Un ejemplo de un sistema de misión crítico es un sistema de navegación para una nave espacial.
3. *Sistemas de negocio críticos.* Son sistemas cuyo fallo de funcionamiento puede provocar costes muy elevados para el negocio que utiliza un sistema de este tipo. Un ejemplo de un sistema de negocio crítico es un sistema de cuentas bancarias.

La propiedad más importante de un sistema crítico es su confiabilidad. El término confiabilidad fue propuesto por Laprie (Laprie, 1995) para hacer referencia a las siguientes propiedades relacionadas de los sistemas: disponibilidad, fiabilidad, seguridad y protección. Tal y como se indica en la Sección 3.2, estas propiedades están enlazadas inextricablemente; por lo tanto, tener un único término para referirse a todas ellas tiene sentido.

Existen varias razones por las que la confiabilidad es la propiedad más importante de los sistemas críticos:

1. *Los sistemas que son no fiables, inseguros o desprotegidos son rechazados a menudo por sus usuarios.* Si los usuarios no confían en un sistema, se negarán a utilizarlo. Es más, también rehusarán comprar o utilizar productos de la misma compañía que produjo el sistema no confiable, puesto que creen que éstos tampoco son confiables.
2. *Los costes de los fallos de funcionamiento del sistema pueden ser enormes.* En algunas aplicaciones, como un sistema de control de reactores o un sistema de navegación aérea, el coste de un fallo en el sistema es mayor en varios órdenes de magnitud que el coste de dicho sistema de control.
3. *Los sistemas no confiables pueden provocar pérdida de información.* Es muy cara la captura y mantenimiento de los datos; algunas veces cuesta más que el sistema informático que los procesa. Se tiene que hacer un gran esfuerzo e invertir mucho dinero para duplicar los datos importantes a fin de protegerlos de cualquier corrupción.

El elevado coste de un fallo de funcionamiento en los sistemas críticos implica que se deben usar métodos y técnicas confiables en su desarrollo. Como consecuencia, los sistemas críticos generalmente se desarrollan utilizando técnicas muy probadas en lugar de técnicas novedosas que no han sido objeto de una extensa experiencia práctica. En vez de utilizar métodos y técnicas novedosas, los desarrolladores de sistemas críticos son conservadores por naturaleza. Prefieren utilizar técnicas antiguas cuyas ventajas e inconvenientes son muy conocidos, en lugar de nuevas técnicas que aparentemente son mejores pero cuyos problemas a largo plazo se desconocen.

Para el desarrollo de sistemas críticos, a menudo se utilizan técnicas de ingeniería del software que por lo general no son rentables. Por ejemplo, los métodos matemáticos formales de desarrollo de software (vistos en el Capítulo 10) han sido usados con éxito para sistemas críticos seguros y protegidos (Hall, 1996; Hall y Chapman, 2002). Una razón por la que se usan estos métodos formales es que ayudan a reducir la cantidad de pruebas requeridas. Para sistemas críticos, los costes de verificación y validación generalmente son muy elevados —más del 50% de los costes totales de desarrollo del sistema.

Si bien un número reducido de sistemas se pueden automatizar completamente, la mayoría de los sistemas críticos son sistemas socio-técnicos en los que las personas monitorizan y controlan el funcionamiento de dichos sistemas informáticos. Los costes de un fallo de funcionamiento de los sistemas críticos generalmente son tan altos que es necesario contar con personal adicional en el sistema que pueda hacer frente a situaciones inesperadas, y que pueda recuperar el funcionamiento normal del sistema cuando las cosas van mal.

Desde luego, a pesar de que los operadores del sistema pueden ayudar a recuperarlo cuando algo va mal, ellos mismos a su vez pueden generar problemas si cometen errores. Existen tres tipos de «componentes de sistemas» susceptibles de generar un fallo en el sistema:

1. El hardware del sistema puede fallar debido a errores en su diseño, también debido a que los componentes fallan como resultado de errores de fabricación, o debido a que dichos componentes han llegado al final de su vida útil.
2. El software del sistema puede fallar debido a errores en su especificación, diseño o implementación.
3. Los operadores del sistema pueden provocar fallos en el sistema debido a un uso incorrecto del mismo. Así como el hardware y el software son cada vez más fiables, hoy en día los fallos debidos a un mal uso del sistema son probablemente la principal causa de fallos de funcionamiento en el sistema.

Estos fallos pueden interrelacionarse. Un componente hardware que deja de funcionar puede implicar que los operadores del sistema tengan que afrontar una situación inesperada y una carga de trabajo adicional. Esto hace que los operadores trabajen en estado de estrés y las personas que sufren estrés a menudo cometen errores. Esto puede ocasionar que el software falle, lo que supone más trabajo para los operadores, más estrés, y así sucesivamente.

Como resultado de todo lo anterior, es particularmente importante que los diseñadores de los sistemas críticos adopten una perspectiva holística del sistema en lugar de centrarse en un único aspecto del mismo. Si el hardware, el software y las formas de utilización del sistema se diseñan de forma separada sin tener en cuenta los puntos débiles potenciales del resto de las partes del sistema, entonces será más probable que los errores se produzcan en las interfaces entre las distintas partes del sistema.

3.1 Un sistema de seguridad crítico sencillo

Hay muchos tipos de sistemas informáticos críticos, desde sistemas de control para dispositivos y maquinarias hasta sistemas de información y comercio electrónico. Éstos podrían ser excelentes casos de estudio para un libro de ingeniería del software, ya que con frecuencia se usan en su desarrollo técnicas avanzadas de ingeniería del software. Sin embargo, comprender estos sistemas puede resultar muy difícil, puesto que es necesario comprender las características y restricciones del dominio de la aplicación en el que operan.

Como consecuencia, el caso de estudio sobre sistemas críticos que uso en varios capítulos en este libro es un sistema médico que simula el funcionamiento del páncreas (un órgano interno). He elegido éste porque todos nosotros tenemos algún conocimiento acerca de problemas médicos y está claro por qué la seguridad y la fiabilidad son tan importantes para este tipo de sistemas. Se pretende que el sistema elegido ayude a las personas que padecen diabetes.

La diabetes es una enfermedad relativamente común en la cual el cuerpo humano no es capaz de producir suficiente cantidad de una hormona llamada insulina. La insulina metaboliza la glucosa en la sangre. El tratamiento convencional de la diabetes comprende inyecciones frecuentes de insulina fabricada genéticamente. Los diabéticos miden sus niveles de azúcar en la sangre usando un medidor externo y calculan la dosis de insulina que deberían inyectarse.

El problema de este tratamiento es que el nivel de insulina en la sangre no depende solamente del nivel de glucosa en la sangre, sino que también depende del momento en el que se inyectó la insulina. Esto puede conducir a niveles muy bajos de glucosa en la sangre (si hay demasiada insulina) o niveles muy altos de azúcar en la sangre (si hay muy poca insulina). Una bajada de azúcar en la sangre constituye, a corto plazo, un problema más serio, ya que puede ocasionar un mal funcionamiento del cerebro de forma temporal y, en última instancia, provocar la inconsciencia y la muerte. A largo plazo, un nivel alto continuado de azúcar en la sangre puede conducir a daños en los ojos, en los riñones, y problemas de corazón.

Los avances de hoy en día en el desarrollo de sensores miniaturizados hacen posible el desarrollo de sistemas de suministro automático de insulina. Estos sistemas monitorizan el nivel de azúcar en la sangre y suministran la dosis adecuada de insulina en el momento en el que se necesita. Los sistemas de suministro de insulina como el mencionado ya existen para el tratamiento de pacientes en hospitales. En el futuro, será posible para muchos diabéticos llevar dichos sistemas de forma permanente adheridos a su cuerpo.

Un sistema de suministro de insulina controlado por software funciona utilizando un microsensor incrustado en el paciente para medir algún parámetro de la sangre que sea proporcional al nivel de azúcar. Dicha información es enviada al controlador de la bomba. El controlador calcula el nivel de azúcar y la cantidad de insulina que se necesita. A continuación envía señales a una bomba miniaturizada para suministrar la insulina a través de una aguja adherida permanentemente en el paciente.

La Figura 3.1 muestra los componentes y la organización de la bomba de insulina. La Figura 3.2 muestra un modelo de flujo de datos que ilustra cómo una entrada de un

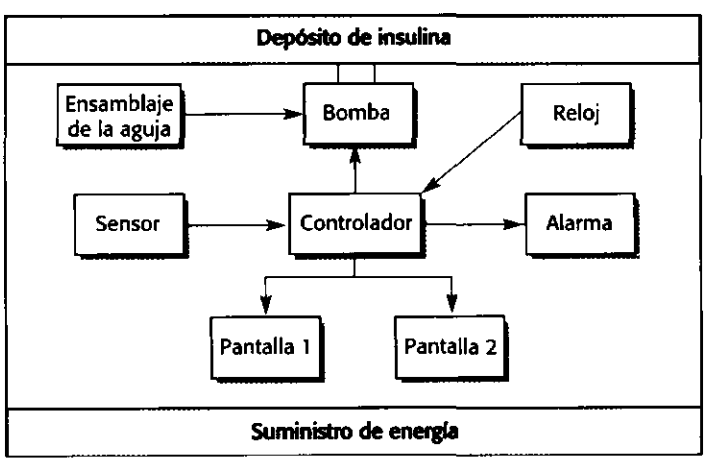


Figura 3.1
Estructura de la bomba de insulina.

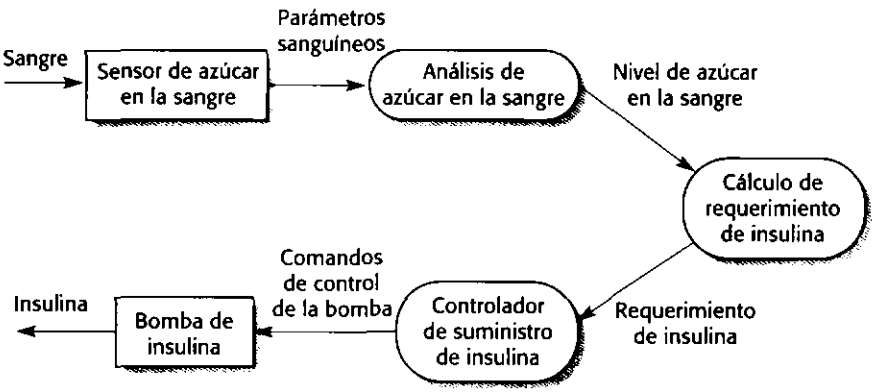


Figura 3.2 Modelo de flujo de datos de la bomba de insulina.

nivel de azúcar en la sangre se transforma en una secuencia de comandos de control de la bomba.

Hay dos requerimientos de alto nivel de confiabilidad para este sistema de bomba de insulina:

1. El sistema deberá estar disponible para suministrar insulina cuando sea necesario.
2. El sistema deberá funcionar de forma fiable y suministrar la cantidad correcta de insulina para contrarrestar el nivel actual de azúcar en la sangre.

Un fallo en el sistema podría, en principio, provocar que se suministren dosis excesivas de insulina, y esto constituiría una amenaza para la vida del paciente. Es particularmente importante que no se produzcan sobredosis de insulina.

3.2 Confiabilidad de un sistema

Todos estamos familiarizados con los problemas derivados de un fallo de funcionamiento en el sistema informático. Por alguna razón no obvia, a veces los sistemas informáticos caen y no consiguen realizar los servicios que se les ha requerido. Los programas que se ejecutan sobre dichos sistemas pueden no funcionar como se esperaba y, ocasionalmente, pueden corromper los datos que son gestionados por el sistema. Hemos aprendido a vivir con este tipo de fallos, y pocos de nosotros confiamos plenamente en las computadoras personales que normalmente usamos.

La confiabilidad de un sistema informático es una propiedad del sistema que es igual a su fidelidad. La fidelidad esencialmente significa el grado de confianza del usuario en que el sistema operará tal y como se espera de él y que no «fallará» al utilizarlo normalmente. Esta propiedad no se puede expresar numéricamente, sino que se utilizan términos relativos como «no confiable», «muy confiable» y «ultraconfiable» para reflejar los grados de confianza que podríamos tener en un sistema.

Grado de confianza y grado de utilidad no son, desde luego, lo mismo. Yo no creo que el procesador de textos que he usado para escribir este libro sea un sistema muy confiable, pero es muy útil. Sin embargo, para reflejar mi falta de confianza en el sistema frecuentemente almaceno mi trabajo y mantengo múltiples copias de seguridad de él. Por lo tanto, compenso la falta de confianza en el sistema con acciones que limitan el daño que podría ocasionarse por una caída del sistema.

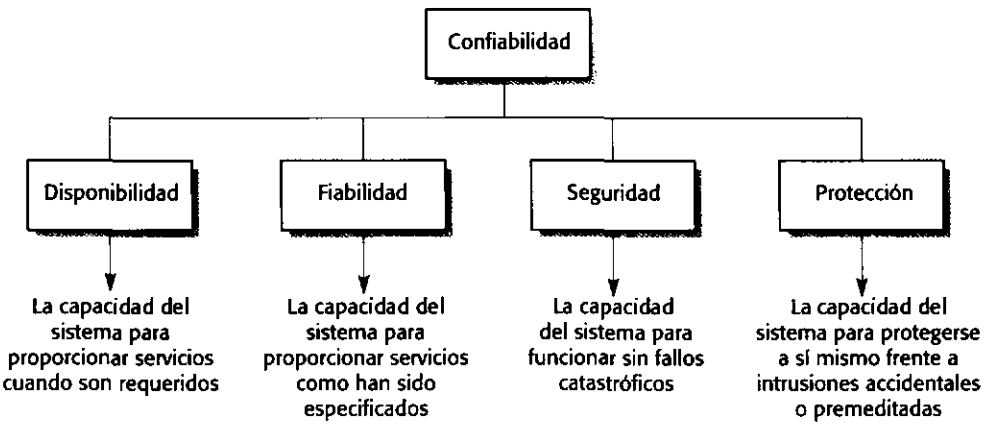


Figura 3.3 Dimensiones de la confiabilidad.

Existen cuatro dimensiones principales de la confiabilidad, tal y como se muestra en la Figura 3.3:

1. *Disponibilidad:* Informalmente, la disponibilidad de un sistema es la probabilidad de que esté activo y en funcionamiento y sea capaz de proporcionar servicios útiles en cualquier momento.
2. *Fiabilidad:* Informalmente, la fiabilidad de un sistema es la probabilidad de que, durante un determinado periodo de tiempo, el sistema funcione correctamente tal y como espera el usuario.
3. *Seguridad:* Informalmente, la seguridad de un sistema es una valoración de la probabilidad de que el sistema cause daños a las personas o a su entorno.
4. *Protección:* Informalmente, la protección de un sistema es una valoración de la probabilidad de que el sistema pueda resistir intrusiones accidentales o premeditadas.

Las propiedades anteriores pueden descomponerse a su vez en otras propiedades más simples. Por ejemplo, la *protección* incluye la *integridad* (asegurar que el programa y los datos de los sistemas no resultan dañados) y la *confidencialidad* (asegurar que sólo las personas autorizadas puedan acceder a la información). La *fiabilidad* incluye la corrección (asegurar que los servicios que proporciona el sistema son los que se han especificado), *precisión* (asegurar que la información se proporciona al usuario con el nivel de detalle adecuado), y *oportunidad* (asegurar que la información que proporciona el sistema se hace cuando es requerida).

Las propiedades de la confiabilidad ya mencionadas de disponibilidad, seguridad, fiabilidad y protección están interrelacionadas. El funcionamiento de un sistema seguro depende normalmente de que el sistema esté disponible y su funcionamiento sea fiable. Un sistema puede convertirse en no fiable debido a que sus datos han sido corrompidos por algún intruso. Los ataques de denegación de servicio en un sistema tienen como propósito comprometer su disponibilidad. Si un sistema que ha demostrado ser seguro es infectado por un virus, ya no se le puede suponer un funcionamiento seguro. Estas interrelaciones entre las cuatro propiedades son la razón de introducir la noción de confiabilidad como una propiedad que las engloba.

Además de estas cuatro dimensiones principales, también se pueden considerar otras propiedades del sistema incluidas en el término confiabilidad:

1. *Reparabilidad.* Los fallos de funcionamiento del sistema son inevitables, pero la interrupción causada por estos fallos se puede minimizar si el sistema se puede repa-

rar rápidamente. Para que esto ocurra, debe ser posible diagnosticar el problema, acceder al componente que ha fallado y realizar los cambios para reparar ese componente. La reparabilidad del software se mejora cuando se tiene acceso al código fuente y se tiene personal con destreza para realizar cambios sobre él. Desgraciadamente, esto es cada vez menos frecuente a medida que nos movemos hacia el desarrollo de sistemas que utilizan componentes comprados a terceros o cajas negras (*véase* el Capítulo 19).

2. *Mantenibilidad.* A medida que se usan los sistemas, surgen nuevos requerimientos. Es importante mantener la utilidad de un sistema cambiándolo para adaptarlo a estos nuevos requerimientos. Un software mantenible es un software que puede adaptarse para tener en cuenta los nuevos requerimientos con un coste razonable y con una baja probabilidad de introducir nuevos errores en el sistema al realizar los cambios correspondientes.
3. *Supervivencia.* Una característica muy importante de los sistemas basados en Internet es la supervivencia, que está estrechamente relacionada con la seguridad y la disponibilidad (Ellison *et al.*, 1999). La supervivencia es la capacidad de un sistema para continuar ofreciendo su servicio mientras está siendo atacado y, potencialmente, mientras parte del sistema está inhabilitado. Las tareas de supervivencia se centran en la identificación de componentes del sistema claves y en asegurar que éstos pueden ofrecer un servicio de funcionamiento mínimo. Se utilizan tres estrategias para asegurar que el sistema pueda continuar funcionando con un servicio mínimo, a saber: resistencia al ataque, reconocimiento del ataque y recuperación de daños ocasionados por un ataque (Ellison *et al.*, 1999; Ellison *et al.*, 2002).
4. *Tolerancia a errores.* Esta propiedad puede considerarse como parte de la usabilidad (explicada en el Capítulo 16) y refleja hasta qué punto el sistema ha sido diseñado para evitar y tolerar un error en la entrada de datos del usuario al sistema. Cuando se producen errores por parte del usuario, el sistema debería, en la medida de lo posible, detectar estos errores y repararlos de forma automática o pedir al usuario que vuelva a introducir sus datos.

Debido a que la disponibilidad, fiabilidad, seguridad y protección son propiedades fundamentales de la confiabilidad, me he centrado en ellas en este capítulo y en posteriores capítulos que tratan la especificación de sistemas críticos (Capítulo 9), desarrollo de sistemas críticos (Capítulo 20) y validación de sistemas críticos (Capítulo 24).

Desde luego, estas propiedades de confiabilidad no son aplicables a todos los sistemas. Para el sistema de bomba de insulina, presentado en la Sección 3.1, las propiedades más importantes son disponibilidad (debe estar en funcionamiento cuando sea requerido), fiabilidad (debe suministrar la dosis correcta de insulina) y seguridad (nunca debe suministrar una dosis peligrosa de insulina). La protección, en este caso, es menos probable que sea una cuestión clave, ya que la bomba no mantiene información confidencial y no está conectada a la red, por lo que no puede ser atacada de forma maliciosa.

Los diseñadores normalmente deben buscar un equilibrio entre el rendimiento del sistema y su confiabilidad. Por lo general, niveles altos de confiabilidad solamente pueden alcanzarse a costa del rendimiento del sistema. Un software confiable incluye código extra, a menudo redundante, para realizar las comprobaciones necesarias para estados excepcionales del sistema y para recuperar el sistema ante un fallo. Esto reduce la confiabilidad del sistema e incrementa la cantidad de memoria requerida por el software. Además, también se incrementan de forma significativa los costes del desarrollo del sistema.

Debido al diseño adicional, implementación y costes de validación, el incremento de la confiabilidad de un sistema puede hacer crecer significativamente los costes de desarrollo. En particular, los costes de validación son elevados para los sistemas críticos. Además de validar que el sistema cumple con sus requerimientos, el proceso de validación tiene que comprobar que el sistema es confiable a través de un sistema de regulación externo, como por ejemplo la Autoridad Federal de Aviación.

La Figura 3.4 muestra la relación entre los costes y las mejoras crecientes en la confiabilidad. Cuanto mayor sea la confiabilidad que se necesita, más habrá que gastar en probar y chequear que efectivamente se ha alcanzado dicho nivel de confiabilidad. Debido al carácter exponencial de esta curva coste/confiabilidad, no es posible demostrar que un sistema es totalmente confiable, ya que los costes necesarios para asegurar esto podrían ser infinitos.

3.3 Disponibilidad y fiabilidad

La disponibilidad y fiabilidad de un sistema son propiedades que están estrechamente relacionadas y que pueden expresarse como probabilidades numéricas. La fiabilidad de un sistema es la probabilidad de que el sistema funcione correctamente tal y como se ha especificado. La disponibilidad de un sistema es la probabilidad de que el sistema esté en disposición de funcionar para proporcionar los servicios a los usuarios que lo soliciten.

Si bien estas dos propiedades guardan una estrecha relación, no se puede deducir que los sistemas fiables estarán siempre disponibles y viceversa. Por ejemplo, algunos sistemas pueden tener como requisito una disponibilidad alta, pero una fiabilidad mucho más baja. Si los usuarios esperan un servicio continuo, entonces los requerimientos de disponibilidad son altos. Sin embargo, si las consecuencias de un fallo de funcionamiento son mínimas y el sistema puede recuperarse rápidamente de dichos fallos, entonces el mismo sistema puede tener requerimientos de fiabilidad bajos.

Un ejemplo de un sistema en el que la disponibilidad es más crítica que la fiabilidad es una centralita telefónica. Los usuarios esperan escuchar un tono cuando descuelgan el teléfono, de forma que el sistema requiere un alto nivel de disponibilidad. Sin embargo, si un defecto en el sistema hace que la conexión termine, ésta es a menudo recuperable. Los conmutadores

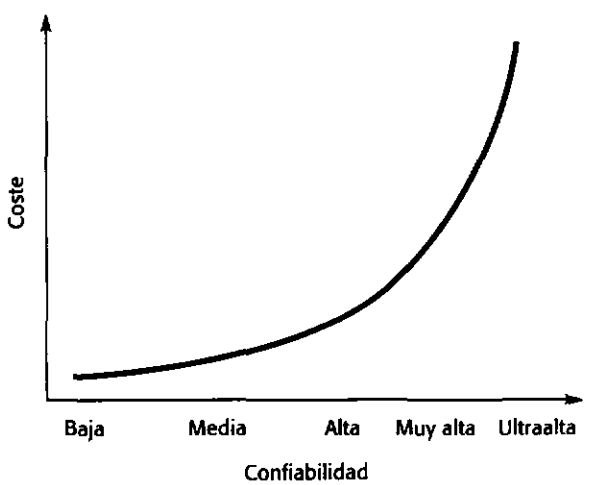


Figura 3.4 Curva de coste/confiabilidad.

de las centralitas normalmente incluyen facilidades para reiniciar el sistema y volver a intentar establecer la conexión. Esto puede realizarse de forma muy rápida, y el usuario puede incluso no darse cuenta de que ha ocurrido un fallo de funcionamiento. Por lo tanto, la disponibilidad es el requerimiento clave en estos sistemas en vez de la fiabilidad.

Una diferencia adicional entre estas características es que la disponibilidad no depende simplemente del sistema en sí, sino también del tiempo necesario para reparar los defectos que hicieron que el sistema dejara de estar disponible. Por ello, si un sistema A falla una vez al año, y un sistema B falla una vez al mes, entonces A claramente es más fiable que B. Sin embargo, supóngase que el sistema A tarda tres días en recuperarse después de un fallo, mientras que B tarda 10 minutos en reinicializarse. La disponibilidad de B a lo largo del año (120 minutos de tiempo sin servicio) es mucho mejor que la del sistema A (4.230 minutos sin servicio).

La fiabilidad y la disponibilidad de un sistema se pueden definir de forma más precisa como sigue:

1. *Fiabilidad.* La probabilidad de que se tengan operaciones libres de caídas durante un tiempo definido en un entorno dado para un propósito específico.
2. *Disponibilidad.* La probabilidad de que un sistema, en cierto momento, esté en funcionamiento y sea capaz de proporcionar los servicios solicitados.

Uno de los problemas prácticos en el desarrollo de sistemas fiables es que nuestras nociones intuitivas de fiabilidad y disponibilidad son a menudo más amplias que estas definiciones limitadas. La definición de *fiabilidad* establece que debe tenerse en cuenta el entorno en el que el sistema se utiliza y el propósito para el que se utiliza. Si se mide la fiabilidad del sistema en un entorno, no se puede suponer que la fiabilidad será la misma en otro entorno en el que el sistema se usa de forma diferente.

Por ejemplo, consideremos que medimos la fiabilidad de un procesador de texto en un entorno de oficina en el que la mayoría de los usuarios no están interesados en cómo funciona. Ellos siguen las instrucciones para su uso y no tratan de experimentar con el sistema. Si se mide la fiabilidad del mismo sistema en un entorno universitario, entonces la fiabilidad puede ser bastante diferente. Aquí, los estudiantes pueden explorar los límites del sistema y usarlo de formas inesperadas. Esto puede producir fallos de funcionamiento en el sistema que no ocurrirían en un entorno de oficina más restringido.

Las percepciones y patrones humanos de utilización también son significativos. Por ejemplo, consideremos una situación en la que el limpiaparabrisas de un automóvil tiene un defecto, lo que hace que los limpiadores no funcionen correctamente bajo una tormenta. La fiabilidad de este sistema percibida por el conductor depende de dónde viva éste y el uso que se le dé al automóvil. Un conductor de Seattle (clima lluvioso) probablemente se verá más afectado por este fallo que un conductor de Las Vegas (clima seco). La percepción del conductor de Seattle será que el sistema no es fiable, mientras que el conductor de Las Vegas puede que nunca se dé cuenta del problema.

Una dificultad adicional con estas definiciones es que no tienen en cuenta ni la gravedad de los fallos ni las consecuencias de la ausencia de disponibilidad. Las personas, naturalmente, se preocupan más por los fallos del sistema que tienen consecuencias graves, y su percepción de la fiabilidad se verá influenciada por dichas consecuencias. Por ejemplo, consideremos un fallo en el software que controla un motor de coche que hace que se cale inmediatamente después de encenderlo, pero dicho motor funciona correctamente después de volver a arrancarlo, corrigiendo el problema de arranque. Esto no afecta al funcionamiento normal del coche, y muchos conductores no pensarán en que es necesaria una reparación.

Por el contrario, la mayoría de los conductores pensarán que un motor que se cale una vez al mes mientras estén conduciendo a alta velocidad (por ejemplo) no es fiable ni seguro y debe ser reparado.

Una definición estricta de fiabilidad relaciona la implementación del sistema con su especificación. Esto es, el sistema funciona de forma fiable si su comportamiento es consecuente con el definido en la especificación. Sin embargo, una razón habitual por la que se advierte una falta de fiabilidad es que la especificación del sistema no cumpla con las expectativas de los usuarios del sistema. Por desgracia, muchas especificaciones son incompletas o incorrectas, y se deja a los ingenieros de software que interpreten cómo debería funcionar el sistema. Debido a que ellos no son expertos en el dominio, no pueden, por lo tanto, implementar el comportamiento que los usuarios esperan.

La fiabilidad y la disponibilidad están relacionadas con los fallos de funcionamiento del sistema. Éstos pueden ser un fallo al proporcionar un servicio, un fallo provocado por la forma en que se proporciona dicho servicio, o la prestación de un servicio de modo que éste sea inseguro o no protegido. Algunos de estos fallos son consecuencia de errores de especificación o fallos en sistemas asociados, como los sistemas de telecomunicaciones. Sin embargo, muchos fallos son consecuencia de comportamientos erróneos del sistema derivados de defectos existentes en éste. Cuando se analiza la fiabilidad, es útil distinguir entre los términos defecto, error y fallo. Estos términos se han definido en la Figura 3.5.

Los errores humanos no conducen de forma inevitable a fallos de funcionamiento del sistema. Los defectos introducidos pueden estar en partes del sistema que nunca han sido usadas. Los defectos no conducen necesariamente a errores del sistema, ya que el estado defectuoso puede ser transitorio y puede corregirse antes de que tenga lugar el comportamiento erróneo. Los errores del sistema pueden no provocar fallos de funcionamiento del sistema, ya que el comportamiento puede ser también transitorio y puede tener efectos inapreciables o el sistema puede incluir protección que asegure que el comportamiento erróneo sea descubierto y corregido antes de que el funcionamiento del sistema se vea afectado.

La diferencia entre los términos mostrados en la Figura 3.5 nos ayuda a identificar tres enfoques complementarios usados para mejorar la fiabilidad de un sistema:

1. *Evitación de defectos*. Se utilizan técnicas que minimizan la posibilidad de cometer equivocaciones y/o detectan las equivocaciones antes de que provoquen la introducción de defectos en el sistema. Ejemplos de tales técnicas son evitar el empelo de construcciones de lenguajes de programación propensas a errores, como los punteros, y el uso de análisis estático para detectar anomalías en el programa.

Fallo del sistema (en inglés, *system failure*)	**Evento que tiene lugar en algún instante cuando el sistema no funciona como esperan sus usuarios.**
Error del sistema (en inglés, *system error*)	**Estado erróneo del sistema que puede dar lugar a un comportamiento del mismo inesperado por sus usuarios.**
Defecto del sistema (en inglés, *system fault*)	**Característica de un sistema software que puede dar lugar a un error del sistema. Por ejemplo, un fallo en la ejecución al inicializar una variable puede hacer que dicha variable tenga un valor incorrecto cuando sea usada.**
Error humano o equivocación (en inglés, *mistake*)	**Comportamiento humano que tiene como consecuencia la introducción de defectos en el sistema.**

Figura 3.5 Terminología de la fiabilidad.

2. *Detección y eliminación de defectos.* Se usan técnicas de verificación y validación que incrementan la posibilidad de que los defectos se detecten y eliminen antes de utilizar el sistema. Un ejemplo son las pruebas sistemáticas del sistema y la depuración.
3. *Tolerancia a defectos.* Se usan técnicas que aseguran que los defectos en un sistema no conducen a errores del sistema o que aseguran que los errores del sistema no dan lugar a fallos de funcionamiento del sistema. Ejemplos de dichas técnicas son la incorporación de facilidades de autodetección en un sistema y el uso de módulos redundantes del sistema.

El desarrollo de sistemas tolerantes a defectos se trata en el Capítulo 20, en donde se explican algunas técnicas para prevención de defectos. En el Capítulo 27 se estudian las aproximaciones basadas en procesos para la prevención de defectos y la detección de defectos se trata en los Capítulos 22 y 23.

Los defectos del software ocasionan fallos de funcionamiento del software cuando el código con defectos se ejecuta con un conjunto de entradas que ponen de manifiesto los defectos del software. El código funciona correctamente para la mayoría de las entradas. La Figura 3.6, obtenida de Littlewood (Littlewood, 1990), muestra un sistema software como una correspondencia entre un conjunto de entradas y un conjunto de salidas. Dada una entrada o una secuencia de entradas, el programa responde produciendo la correspondiente salida. Por ejemplo, dada una entrada de una URL, un navegador web produce una salida que es la pantalla de la página web solicitada.

Algunas de estas combinaciones de entradas o salidas, mostradas en la elipse sombreada de la Figura 3.6, provocan la generación de salidas erróneas. La fiabilidad del software está relacionada con la probabilidad de que, en una ejecución particular del programa, la entrada del sistema pertenezca al conjunto de entradas que hacen que se produzca una salida errónea. Si una entrada que ocasiona una salida errónea se asocia con una parte del sistema que se usa con frecuencia, entonces los fallos de funcionamiento del sistema serán frecuentes. Sin embargo, si se asocia con código que se usa raramente, entonces los usuarios difícilmente verán los fallos de funcionamiento.

Cada usuario de un sistema lo usa de diferentes formas. Los defectos que afectan a la fiabilidad del sistema para un usuario puede que nunca se manifiesten bajo otro modo de

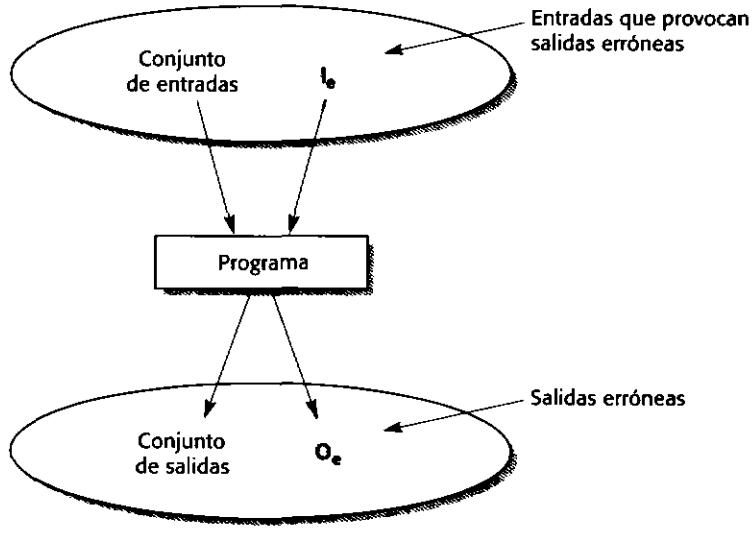


Figura 3.6 Un sistema visto como una correspondencia entre entradas y salidas.

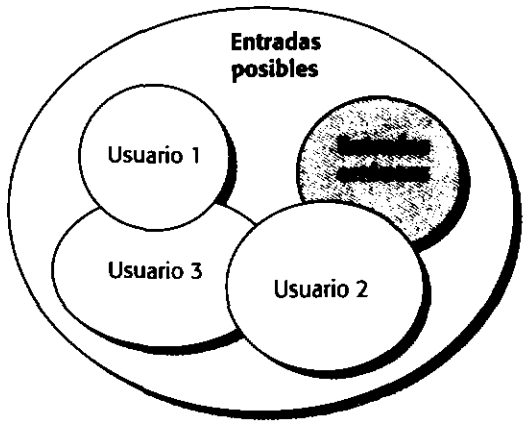


Figura 3.7 OJO falta traducción. Software usage patterns.

trabajo (Figura 3.7). En la Figura 3.7, el conjunto de entradas erróneas se corresponden con la elipse sombreada de la Figura 3.6. El conjunto de entradas producido por el Usuario 2 se intersecta con este conjunto de entradas erróneas. El Usuario 2, por tanto, experimentará algunos fallos de funcionamiento del sistema. El Usuario 1 y el Usuario 3, sin embargo, nunca usarán entradas del conjunto erróneo. Para ellos, el software siempre será fiable.

La fiabilidad de un programa, por lo tanto, depende principalmente del número de entradas que provocan salidas erróneas durante el uso normal del sistema llevado a cabo por la mayoría de usuarios. Los defectos del software que se manifiestan solamente en situaciones excepcionales tienen poco efecto en la fiabilidad del sistema. La eliminación de defectos del software en partes de un sistema que se utilizan raramente, hace que haya poca diferencia real con la fiabilidad percibida por los usuarios del sistema. Mills y otros (Mills *et al.*, 1987) señalan que la eliminación del 60% de errores conocidos en su software permite una mejora de la fiabilidad solamente del 3%. Adams (Adams, 1984), en un estudio de productos software de IBM, observó que muchos defectos probablemente provocarán fallos de ejecución después de cientos o miles de meses de utilizar el producto.

Los usuarios en un sistema socio-técnico pueden adaptarse al software con defectos conocidos, y pueden compartir información acerca de cómo esquivar dichos problemas. Pueden evitar el uso de entradas que saben que producirán problemas, por lo que los fallos de funcionamiento nunca tendrán lugar. Además, los usuarios experimentados suelen eludir los defectos del software que saben que provocarán fallos de funcionamiento. Evitan de forma deliberada usar funcionalidades del sistema que saben que pueden causarles problemas. Por ejemplo, yo evito usar ciertas funcionalidades, como la numeración automática con el procesador de textos que he usado para escribir este libro. La reparación de defectos en estas funcionalidades puede no producir prácticamente ninguna diferencia con la fiabilidad percibida por dichos usuarios.

3.4 Seguridad

Los sistemas de seguridad críticos son sistemas en los que es esencial que el funcionamiento del sistema sea siempre seguro. Esto es, el sistema *nunca* debería provocar daños en las personas o en el entorno del sistema incluso si éste falla. Ejemplos de sistemas de

seguridad críticos son el control y monitorización de sistemas de un avión, sistemas de control de procesos en plantas químicas y farmacéuticas y sistemas de control de automóviles.

El control mediante hardware de los sistemas de seguridad críticos es más sencillo de implementar y analizar que el control mediante software. Sin embargo, actualmente se están construyendo sistemas de tal complejidad que no se pueden controlar únicamente mediante hardware. Es esencial realizar algún control mediante software debido a la necesidad de gestionar un número muy grande de sensores y actuadores con leyes de control complejas. Un ejemplo que muestra dicha complejidad se encuentra en los aviones militares avanzados inestables aerodinámicamente. Dichos sistemas requieren ajustes continuos, controlados por software, de sus superficies de vuelo para asegurar que no se estrellen.

El software de seguridad crítico se divide en dos clases:

1. *Software de seguridad crítico primario.* Es el software que está embebido como un controlador en un sistema. El mal funcionamiento de dicho software puede ocasionar un mal funcionamiento del hardware, lo que puede provocar lesiones personales o daños en el entorno. Aquí nos centramos en este tipo de software.
2. *Software de seguridad crítico secundario.* Es el software que indirectamente puede provocar lesiones. Ejemplos de dichos sistemas son los sistemas de diseño asistido por computadora, cuyo mal funcionamiento podría provocar un defecto de diseño en el objeto que se está diseñando. Este defecto puede causar lesiones personales si el sistema diseñado no funciona bien. Otro ejemplo de un sistema de seguridad crítico secundario es una base de datos médica que contiene los detalles de los medicamentos administrados a los pacientes. Los errores en este sistema podrían dar lugar a que se administrara una dosis de medicamentos incorrecta.

La fiabilidad y la seguridad del sistema están relacionadas, pero son distintos atributos de confiabilidad. Desde luego, un sistema de seguridad crítico es fiable si está de acuerdo con su especificación y funciona sin fallos. Dicho sistema puede incorporar características de tolerancia a defectos para que pueda proporcionar un servicio continuo incluso si se producen defectos. Sin embargo, los sistemas tolerantes a defectos no son necesariamente seguros. El software aún puede funcionar mal y ocasionar un comportamiento del sistema que provoque un accidente.

Además del hecho de que nunca podemos tener la certeza absoluta de que un sistema software está libre de defectos y es tolerante a fallos, hay muchas otras razones por las que un sistema software que es fiable no necesariamente es seguro:

1. La especificación puede estar incompleta en el sentido de que no describe el comportamiento requerido del sistema en algunas situaciones críticas. Un alto porcentaje de sistemas que funcionan mal (Natajo y Kume, 1991; Lutz, 1993) se debe a errores de especificación más que a errores de diseño. En un estudio de errores en sistemas empotrados, Lutz concluye que:

   ... las dificultades con los requerimientos son la causa clave de los errores de software relacionados con la seguridad que persistieron hasta la integración y la prueba del sistema.

2. El mal funcionamiento del hardware hace que el sistema se comporte de forma impredecible y enfrente al software con un entorno inesperado. Cuando los componentes están próximos a fallar, pueden comportarse de forma errática y generar señales que están fuera de los rangos que puede manejar el software.

3. Los operadores del sistema pueden generar entradas que no son individualmente incorrectas, pero que, en situaciones particulares, pueden dar lugar a un mal funcionamiento del sistema. Como ejemplo anecdótico se puede citar el caso en que un mecánico dio instrucciones al software de utilidades de gestión de un avión para que levantara el tren de aterrizaje. El software ejecutó las instrucciones perfectamente. Por desgracia, el avión permaneció en tierra todo el tiempo; claramente, el sistema debería haber inhabilitado el comando a menos que el avión estuviese en el aire.

Se ha creado un vocabulario especializado para tratar los sistemas de seguridad críticos y es importante comprender los términos específicos utilizados. La Figura 3.8 recoge algunas definiciones que he adaptado de los términos inicialmente definidos por Leveson (Leveson, 1985).

La clave para garantizar la seguridad es asegurar que los accidentes no ocurran o que las consecuencias de éstos sean mínimas. Esto puede conseguirse de tres formas complementarias:

1. *Evitación de contingencias.* El sistema se diseña para que las contingencias se eviten. Por ejemplo, un sistema de corte que requiere que el operador presione dos botones distintos al mismo tiempo para utilizar la máquina evita la contingencia de que los dedos del operador estén cerca de las cuchillas.
2. *Detección y eliminación de contingencias.* El sistema se diseña para que las contingencias se detecten y eliminen antes de que provoquen un accidente. Por ejemplo, un sistema de una planta química puede detectar una presión excesiva y abrir una válvula de escape para reducir la presión antes de que ocurra una explosión.
3. *Limitación de daños.* El sistema incluye características de protección que minimizan el daño que puede resultar de un accidente. Por ejemplo, el motor de un avión normalmente incluye extintores de incendios automáticos. Si se produce un fuego, a menudo éste se puede controlar antes de que suponga una amenaza para el avión.

Los accidentes ocurren generalmente cuando varias cosas van mal al mismo tiempo. Un análisis de accidentes serios (Perrow, 1984) sugiere que casi todos ellos se debieron a una

Accidente (o percance)	**Evento o secuencia de eventos no planificados que provocan muerte o lesiones, daño a las propiedades o al entorno. Un ejemplo de un accidente es una máquina controlada por un ordenador que lesiona a su operador.**
Contingencia	**Una condición con el potencial de causar o contribuir a un accidente. Un ejemplo de contingencia es un fallo de funcionamiento de un sensor que detecta un obstáculo delante de una máquina.**
Daño	**Medida de la pérdida resultante de un percance. El daño puede variar desde varias personas muertas como resultado de un accidente, hasta lesiones o daños menores a la propiedad.**
Gravedad de la contingencia	**Evaluación del peor daño posible que podría resultar de una contingencia en particular. La gravedad de la contingencia puede variar desde catastrófica, en donde muchas personas mueren, a menor, en donde resultan solamente daños menores.**
Probabilidad de la contingencia	**La probabilidad de la ocurrencia de eventos que provocan una contingencia. Los valores de probabilidad tienden a ser arbitrarios, pero varían desde *probable* (por ejemplo, una probabilidad de ocurrencia del 1/100) hasta *improbable* (situaciones no concebibles en las que probablemente ocurra la contingencia).**
Riesgo	**Es una medida de la probabilidad de que el sistema provoque un accidente. El riesgo se evalúa considerando la probabilidad de la contingencia, la gravedad de la contingencia y la probabilidad de que una contingencia cause un accidente.**

Figura 3.8 Terminología sobre seguridad.

combinación de malos funcionamientos más que a fallos aislados. La combinación no anticipada condujo a interacciones que provocaron fallos de funcionamiento del sistema. Perrow sugiere también que es imposible anticiparse a todas las posibles combinaciones de mal funcionamiento de un sistema, y que los accidentes son una parte inevitable del uso de sistemas complejos. El software tiende a incrementar la complejidad del sistema, de forma que realizar el control mediante software puede incrementar la probabilidad de accidentes del sistema.

Sin embargo, el software de control y monitorización puede mejorar también la seguridad de los sistemas. Los sistemas controlados por software pueden monitorizar un rango de condiciones más amplio que los sistemas electromecánicos. Los primeros se pueden adaptar con relativa facilidad. Además implican el uso del hardware de la computadora, el cual tiene una fiabilidad inherente muy alta y es físicamente pequeño y ligero. Los sistemas controlados por software pueden proporcionar mecanismos de seguridad sofisticados. Pueden soportar estrategias de control que reducen la cantidad de tiempo que las personas necesitan consumir en entornos con contingencias. En consecuencia, si bien el software de control puede introducir más formas en las que un sistema puede funcionar mal, también permite una mejor monitorización y protección y, por lo tanto, puede mejorar la seguridad del sistema.

En todos los casos, es importante mantener un sentido de la proporción sobre la seguridad del sistema. Es imposible conseguir que un sistema sea totalmente seguro, y la sociedad debe decidir si los beneficios del uso de tecnologías avanzadas compensan o no las consecuencias de un accidente ocasional. También es una decisión social y política cómo utilizar unos recursos nacionales limitados a fin de reducir el riesgo para la población en su conjunto.

3.5 Protección

La protección es un atributo del sistema que refleja su capacidad para protegerse de ataques externos que pueden ser accidentales o provocados. La protección ha adquirido cada vez más importancia en tanto que más y más sistemas se han conectado a Internet. Las conexiones a Internet proporcionan funcionalidades del sistema adicionales (por ejemplo, los clientes pueden acceder directamente a sus cuentas bancarias), pero la conexión a Internet también significa que el sistema puede ser atacado por personas con intenciones hostiles. La conexión a Internet también conlleva que los detalles sobre vulnerabilidades particulares del sistema pueden difundirse fácilmente para que más personas sean capaces de atacar al sistema. Del mismo modo, sin embargo, la conexión puede acelerar la distribución de parches del sistema para reparar estas vulnerabilidades.

Ejemplos de ataques podrían ser los virus, el uso no autorizado de servicios del sistema y la modificación no autorizada del sistema o sus datos. La protección es importante para todos los sistemas críticos. Sin un nivel razonable de protección, la disponibilidad, fiabilidad y seguridad del sistema pueden verse comprometidas si ataques externos provocan daños al mismo.

La razón de esto es que todos los métodos para asegurar la disponibilidad, fiabilidad y seguridad se valen del hecho de que el sistema operacional es el mismo que se instaló originalmente. Si dicho sistema instalado se ha visto comprometido de alguna forma (por ejemplo, si el software se ha modificado para aceptar un virus), entonces los argumentos para la fiabilidad y la seguridad originalmente establecidos dejan de ser ciertos. El sistema de software puede entonces corromperse y comportarse de forma impredecible.

Por el contrario, los errores en el desarrollo de un sistema pueden provocar agujeros de protección. Si un sistema no responde a entradas inesperadas o si los límites de un vector no se verifican, entonces los atacantes pueden explotar estas debilidades para tener acceso al sistema. Los inci-

dentes de protección más importantes tales como el gusano de Internet original (Spafford, 1989) y el gusano *Code Red* más de diez años después (Berghel, 2001) se aprovecharon del hecho de que los programas en C no incluyen verificación de los límites de los vectores. Los gusanos sobrescribieron parte de la memoria con código que permitió el acceso no autorizado al sistema.

Por supuesto, en algunos sistemas críticos, la protección es la dimensión más importante de la confiabilidad del sistema. Los sistemas militares, los sistemas de comercio electrónico y los sistemas que implican el procesamiento e intercambio de información confidencial, se deben diseñar de tal forma que alcancen altos niveles de protección. Si un sistema de reservas de billetes de avión (por ejemplo) no está disponible, esto provoca inconvenientes y algunos retrasos en la emisión de los billetes. Sin embargo, si el sistema no está protegido y puede aceptar reservas falsas, entonces la línea aérea propietaria del sistema puede perder una gran cantidad de dinero.

Existen tres tipos de daños que pueden ser causados por ataques externos:

1. *Denegación de servicio.* El sistema puede verse forzado a entrar en un estado en que sus servicios normales no están disponibles. Esto, obviamente, afecta a la disponibilidad del sistema.
2. *Corrupción de programas o datos.* Los componentes software del sistema pueden ser alterados de forma no autorizada. Esto puede afectar al comportamiento del sistema y, por lo tanto, a su fiabilidad y a su seguridad. Si el daño es grave, la disponibilidad del sistema puede verse afectada.
3. *Revelación de información confidencial.* La información gestionada por el sistema puede ser confidencial y los ataques externos pueden exponerla a personas no autorizadas. Dependiendo del tipo de datos, esto podría afectar a la seguridad del sistema y puede permitir ataques posteriores que afecten a la disponibilidad o fiabilidad del sistema.

Como con otros aspectos de la confiabilidad, existe una terminología especializada asociada con la protección. Algunos términos importantes, como los tratados por Pfleeger (1977), se definen en la Figura 3.9.

Existe una clara analogía con cierta terminología de la seguridad en el sentido de que una exposición es análoga a un accidente y una vulnerabilidad es análoga a una contingencia. Por tanto, existen aproximaciones comparables que se utilizan para garantizar la protección de un sistema:

1. *Evitar la vulnerabilidad.* El sistema se diseña para que las vulnerabilidades no ocurran. Por ejemplo, si un sistema no está conectado a una red pública externa, entonces no existe la posibilidad de un ataque por parte de otras personas conectadas a la red.

Exposición	**Posible pérdida o daño en un sistema informático. Un ejemplo puede ser la pérdida o daño de los datos o la pérdida de tiempo y esfuerzo si es necesaria una recuperación del sistema después de una violación de protección.**
Vulnerabilidad	**Debilidad en un sistema informático que se puede aprovechar para provocar pérdidas o daños.**
Ataque	**Aprovechamiento de la vulnerabilidad de un sistema. Generalmente, se produce desde fuera del sistema y con una intención deliberada de causar algún daño.**
Amenazas	**Circunstancias que potencialmente pueden provocar pérdidas o daños. Se pueden entender como una vulnerabilidad del sistema que está expuesto a un ataque.**
Control	**Medida de protección que reduce la vulnerabilidad del sistema. La encriptación podría se un ejemplo de un control que reduce una vulnerabilidad de un sistema de control de acceso deficiente.**

Figura 3.9 Terminología sobre protección.

2. *Detección y neutralización de ataques.* El sistema se diseña para detectar vulnerabilidades y eliminarlas antes de que provoquen una exposición del sistema. Un ejemplo de detección y eliminación de la vulnerabilidad es la utilización de un verificador de virus que analiza los ficheros entrantes y los modifica para eliminar el virus.
3. *Limitación de la exposición.* Las consecuencias de un ataque exitoso se minimizan. Ejemplos de limitación de la exposición son los sistemas de copias de seguridad periódicas y una política de gestión de configuraciones que permite que el software dañado pueda reconstruirse.

La gran mayoría de las vulnerabilidades en los sistemas informáticos se originan en fallos humanos en lugar de en problemas técnicos. Las personas eligen palabras clave fáciles de recordar o las escriben en lugares en donde resulta fácil encontrarlas. Los administradores del sistema cometen errores en la actualización del control de acceso o de ficheros de configuración y los usuarios olvidan instalar o usar software de protección. Para mejorar la protección, por lo tanto, necesitamos adoptar una perspectiva socio-técnica y pensar en cómo se usan realmente los sistemas y no solamente en sus características técnicas.

PUNTOS CLAVE

- En un sistema crítico, un fallo de funcionamiento puede provocar pérdidas económicas importantes, daños físicos o amenazas a la vida humana. Tres clases importantes de sistemas críticos son los sistemas de seguridad críticos, sistemas de misión críticos y sistemas de negocio críticos.
- La confiabilidad de un sistema informático es una propiedad del sistema que refleja el grado de confianza que el usuario tiene en el sistema. Las dimensiones más importantes de la confiabilidad son la disponibilidad, fiabilidad, seguridad y protección.
- La disponibilidad de un sistema es la probabilidad de que le sea posible entregar los servicios a sus usuarios cuando se lo soliciten y la fiabilidad es la probabilidad de que los servicios del sistema se entreguen de acuerdo con lo especificado.
- La fiabilidad y la disponibilidad se consideran normalmente como las dimensiones más importantes de la confiabilidad. Si un sistema no es fiable, es difícil asegurar la seguridad del sistema o su protección, ya que éstas pueden verse comprometidas por fallos de funcionamiento del sistema.
- La fiabilidad se relaciona con la probabilidad de que se produzca un error en el momento de utilizar el sistema. Un programa puede contener defectos conocidos, pero aún puede considerarse como fiable por sus usuarios. Éstos pueden no usar nunca las características del sistema que están afectadas por esos defectos.
- La seguridad de un sistema es un atributo de éste que refleja la capacidad del sistema para funcionar, de forma normal o anormalmente, sin amenazar a las personas o al entorno.
- La protección es importante para todos los sistemas críticos. Sin un nivel de protección razonable, la disponibilidad, fiabilidad y seguridad de un sistema pueden verse comprometidas si ataques externos provocan algún daño al sistema.
- Para mejorar la confiabilidad, es necesario adoptar una aproximación socio-técnica para el diseño del sistema, teniendo en cuenta a las personas que forman parte del sistema así como el hardware y el software.

LECTURAS ADICIONALES

«The evolution of information assurance». Un artículo excelente que trata la necesidad de proteger la información crítica de accidentes y ataques en una organización. [R. Cummings, *IEEE Computer,* 35 (12), diciembre de 2002.]

Practical Design of Safety-critical Computer Systems. Una revisión general del diseño de sistemas de seguridad críticos que trata cuestiones de seguridad y que considera el concepto de sistemas y no simplemente una perspectiva software. (W. R. Dunn, Reliability Press, 2002.)

Secrets and Lies: Digital Security in a Networked World. Un libro excelente y muy legible sobre protección de computadoras desde un punto de vista socio-técnico. (B. Schneier, 2000, John Wiley & Sons.)

«Survivability: Protecting your critical systems». Una introducción accesible al tema de supervivencia y el porqué de su importancia. (R. Ellison *et al., IEEE Internet Computing,* noviembre-diciembre de 1999.)

Computer-related Risks. Una colección extraída del Internet Risks Forum sobre los incidentes ocurridos en sistemas automatizados. Muestra lo que puede ir mal realmente en relación con la seguridad de sistemas. (P. G. Neumann, 1995, Addison-Wesley.)

EJERCICIOS

3.1 ¿Cuáles son los tres tipos principales de sistemas críticos? Explique las diferencias entre ellos.

3.2 Sugiera seis razones de por qué la confiabilidad es importante en sistemas críticos.

3.3 ¿Cuáles son las dimensiones más importantes de la confiabilidad del sistema?

3.4 ¿Por qué el coste para garantizar la confiabilidad es exponencial?

3.5 Sugiera, justificando sus respuestas, por qué los atributos de confiabilidad son probablemente los más críticos para los sistemas siguientes:

- Un servidor de Internet proporcionado por un ISP con miles de clientes.
- Un escalpelo controlado por computadora usado para practicar incisiones en operaciones quirúrgicas.
- Un sistema de control direccional usado en el lanzamiento de un vehículo espacial.
- Un sistema de gestión de finanzas personales a través de Internet.

3.6 Identifique seis productos de consumo que contengan, o que puedan contener en el futuro, sistemas software de seguridad críticos.

3.7 La fiabilidad y la seguridad son atributos de confiabilidad relacionados pero distintos. Describa la distinción más importante entre estos atributos y explique por qué es posible para un sistema fiable ser inseguro y viceversa.

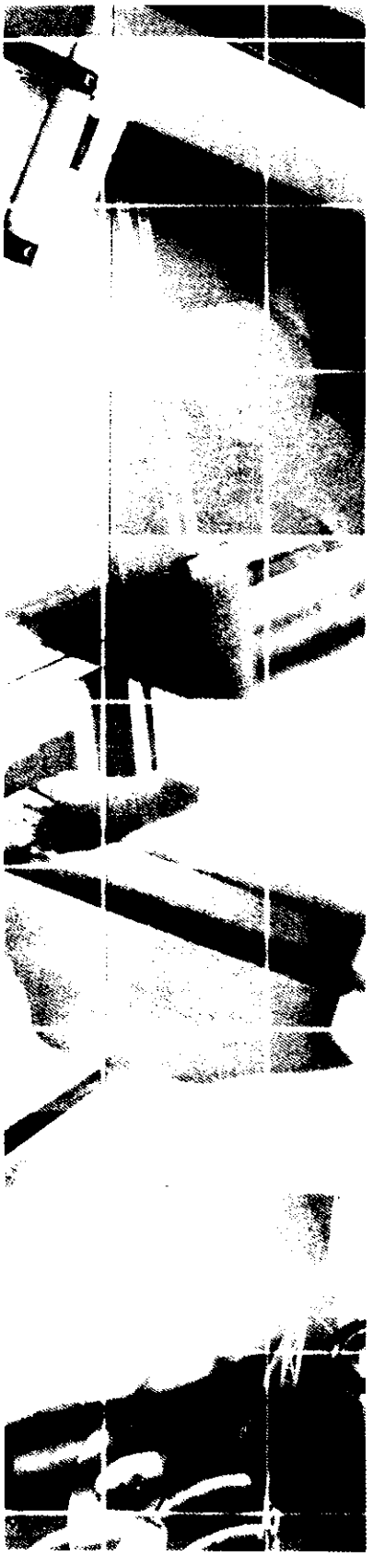
3.8 En un sistema médico diseñado para suministrar radiación para tratar tumores, sugiera una contingencia que pueda ocurrir y proponga una característica del software que pueda usarse para asegurar que la contingencia identificada no derive en un accidente.

3.9 Explique por qué hay una estrecha relación entre disponibilidad del sistema y protección del sistema.

3.10 En términos de protección de computadoras, explique las diferencias entre un ataque y una amenaza.

3.11 ¿Es ético para un ingeniero estar de acuerdo en entregar a un cliente un sistema software con defectos conocidos? ¿Hay alguna diferencia si al cliente se le informa de la existencia de estos defectos con antelación? ¿Podría ser razonable realizar afirmaciones sobre la fiabilidad del software en dichas circunstancias?

3.12 Como experto en protección de computadoras, suponga que una organización que realiza una campaña por los derechos de las víctimas de torturas le pide que ayude a la organización a conseguir accesos no autorizados a los sistemas informáticos de una compañía americana. Esto les ayudaría a confirmar o desmentir que esta compañía está vendiendo equipamiento que se usa directamente en la tortura de prisioneros políticos. Comente los problemas éticos que esta solicitud provoca y cómo podría reaccionar ante esta petición.



4

Procesos del software

Objetivos

El objetivo de este capítulo es introducirlo al concepto del proceso del software —un conjunto coherente de actividades para la producción de software—. Cuando haya leído este capítulo:

- entenderá el concepto de procesos del software y los modelos de estos procesos;
- entenderá tres modelos del proceso del software genéricos y cuándo deben utilizarse;
- entenderá, a grandes rasgos, las actividades relacionadas en la ingeniería de requerimientos del software, desarrollo de software, pruebas y evolución;
- entenderá cómo el Proceso Unificado de Rational integra buenas prácticas del proceso del software para crear un modelo del proceso moderno y genérico;
- habrá sido introducido a la tecnología CASE que se utiliza para ayudar a las actividades del proceso del software.

Contenidos

Un proceso del software es un conjunto de actividades que conducen a la creación de un producto software. Estas actividades pueden consistir en el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, cada vez más, se desarrolla nuevo software ampliando y modificando los sistemas existentes y configurando e integrando software comercial o componentes del sistema.

Los procesos del software son complejos y, como todos los procesos intelectuales y creativos, dependen de las personas que toman decisiones y juicios. Debido a la necesidad de juzgar y crear, los intentos para automatizar estos procesos han tenido un éxito limitado. Las herramientas de ingeniería del software asistida por computadora (CASE) (comentadas en la Sección 4.5) pueden ayudar a algunas actividades del proceso. Sin embargo, no existe posibilidad alguna, al menos en los próximos años, de una automatización mayor en el diseño creativo del software realizado por los ingenieros relacionados con el proceso del software.

Una razón por la cual la eficacia de las herramientas CASE está limitada se halla en la inmensa diversidad de procesos del software. No existe un proceso ideal, y muchas organizaciones han desarrollado su propio enfoque para el desarrollo del software. Los procesos han evolucionado para explotar las capacidades de las personas de una organización, así como las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere un proceso de desarrollo muy estructurado. Para sistemas de negocio, con requerimientos rápidamente cambiantes, un proceso flexible y ágil probablemente sea más efectivo.

Aunque existen muchos procesos diferentes de software, algunas actividades fundamentales son comunes para todos ellos:

1. *Especificación del software*. Se debe definir la funcionalidad del software y las restricciones en su operación.
2. *Diseño e implementación del software*. Se debe producir software que cumpla su especificación.
3. *Validación del software*. Se debe validar el software para asegurar que hace lo que el cliente desea.
4. *Evolución del software*. El software debe evolucionar para cubrir las necesidades cambiantes del cliente.

En este capítulo se tratan brevemente estas actividades y se analizan con más detalle en partes posteriores del libro.

Aunque no existe un proceso del software «ideal», en las organizaciones existen enfoques para mejorarlos. Los procesos pueden incluir técnicas anticuadas o no aprovecharse de las mejores prácticas en la ingeniería del software industrial. De hecho, muchas organizaciones aún no aprovechan los métodos de la ingeniería del software en el desarrollo de su software.

Los procesos del software se pueden mejorar por la estandarización del proceso donde la diversidad de los procesos del software en una organización sea reducida. Esto conduce a mejorar la comunicación y a una reducción del tiempo de formación, y hace la ayuda al proceso automatizado más económica. La estandarización también es un primer paso importante para introducir nuevos métodos, técnicas y buenas prácticas de ingeniería del software. En el Capítulo 28 se trata con más detalle la mejora del proceso del software.

4.1 Modelos del proceso del software

Como se explicó en el Capítulo 1, un modelo del proceso del software es una representación abstracta de un proceso del software. Cada modelo de proceso representa un proceso

desde una perspectiva particular, y así proporciona sólo información parcial sobre ese proceso. En esta sección, se introducen varios modelos de proceso muy generales (algunas veces llamados *paradigmas de proceso*) y se presentan desde una perspectiva arquitectónica. Esto es, vemos el marco de trabajo del proceso, pero no los detalles de actividades específicas.

Estos modelos generales no son descripciones definitivas de los procesos del software. Más bien, son abstracciones de los procesos que se pueden utilizar para explicar diferentes enfoques para el desarrollo de software. Puede pensarse en ellos como marcos de trabajo del proceso que pueden ser extendidos y adaptados para crear procesos más específicos de ingeniería del software.

Los modelos de procesos que se incluyen en este capítulo son:

1. *El modelo en cascada*. Considera las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución, y los representa como fases separadas del proceso, tales como la especificación de requerimientos, el diseño del software, la implementación, las pruebas, etcétera.
2. *Desarrollo evolutivo*. Este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga sus necesidades.
3. *Ingeniería del software basada en componentes*. Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en integrar estos componentes en el sistema más que en desarrollarlos desde cero.

Estos tres modelos de procesos genéricos se utilizan ampliamente en la práctica actual de la ingeniería del software. No se excluyen mutuamente y a menudo se utilizan juntos, especialmente para el desarrollo de sistemas grandes. De hecho, el Proceso Unificado de Rational que se trata en la Sección 4.4 combina elementos de todos estos modelos. Los subsistemas dentro de un sistema más grande pueden ser desarrollados utilizando enfoques diferentes. Por lo tanto, aunque es conveniente estudiar estos modelos separadamente, debe entenderse que, en la práctica, a menudo se combinan.

Se han propuesto todo tipo de variantes de estos procesos genéricos y pueden ser usados en algunas organizaciones. La variante más importante es probablemente el desarrollo formal de sistemas, donde se crea un modelo formal matemático de un sistema. Este modelo se transforma entonces, usando transformaciones matemáticas que preservan su consistencia, en código ejecutable.

El ejemplo más conocido de un proceso de desarrollo formal es el proceso de sala limpia, el cual fue originalmente desarrollado por IBM (Mills *et al.*, 1987; Selby *et al.*, 1987; Linger, 1994; Prowell *et al.*, 1999). En el proceso de sala limpia, cada incremento del software se especifica formalmente, y esta especificación se transforma en una implementación. La corrección del software se demuestra utilizando un enfoque formal. No hay pruebas para defectos en el proceso, y las pruebas del sistema se centran en evaluar su fiabilidad.

Tanto el enfoque de sala limpia como otro enfoque para desarrollo formal basado en el método B (Wordsworth, 1996) son particularmente apropiados para el desarrollo de sistemas que tienen estrictos requerimientos de seguridad, fiabilidad o protección. El enfoque formal simplifica la producción de un caso de seguridad o protección que demuestre a los clientes u organismos de certificación que el sistema realmente cumple los requerimientos de seguridad y protección.

Fuera de estos ámbitos especializados, los procesos basados en transformaciones formales no se utilizan en general. Requieren una pericia especializada y, en realidad, para la mayoría de los sistemas este proceso no ofrece ventajas importantes de coste o calidad sobre otros enfoques para el desarrollo de sistemas.

4.1.1 El modelo en cascada

El primer modelo de proceso de desarrollo de software que se publicó se derivó de procesos de ingeniería de sistemas más generales (Royce, 1970). Este modelo se muestra en la Figura 4.1. Debido a la cascada de una fase a otra, dicho modelo se conoce como modelo en cascada o como ciclo de vida del software. Las principales etapas de este modelo se transforman en actividades fundamentales de desarrollo:

1. *Análisis y definición de requerimientos.* Los servicios, restricciones y metas del sistema se definen a partir de las consultas con los usuarios. Entonces, se definen en detalle y sirven como una especificación del sistema.
2. *Diseño del sistema y del software.* El proceso de diseño del sistema divide los requerimientos en sistemas hardware o software. Establece una arquitectura completa del sistema. El diseño del software identifica y describe las abstracciones fundamentales del sistema software y sus relaciones.
3. *Implementación y prueba de unidades.* Durante esta etapa, el diseño del software se lleva a cabo como un conjunto o unidades de programas. La prueba de unidades implica verificar que cada una cumpla su especificación.
4. *Integración y prueba del sistema.* Los programas o las unidades individuales de programas se integran y prueban como un sistema completo para asegurar que se cumplan los requerimientos del software. Después de las pruebas, el sistema software se entrega al cliente.
5. *Funcionamiento y mantenimiento.* Por lo general (aunque no necesariamente), ésta es la fase más larga del ciclo de vida. El sistema se instala y se pone en funcionamiento práctico. El mantenimiento implica corregir errores no descubiertos en las etapas anteriores del ciclo de vida, mejorar la implementación de las unidades del sistema y resaltar los servicios del sistema una vez que se descubren nuevos requerimientos.

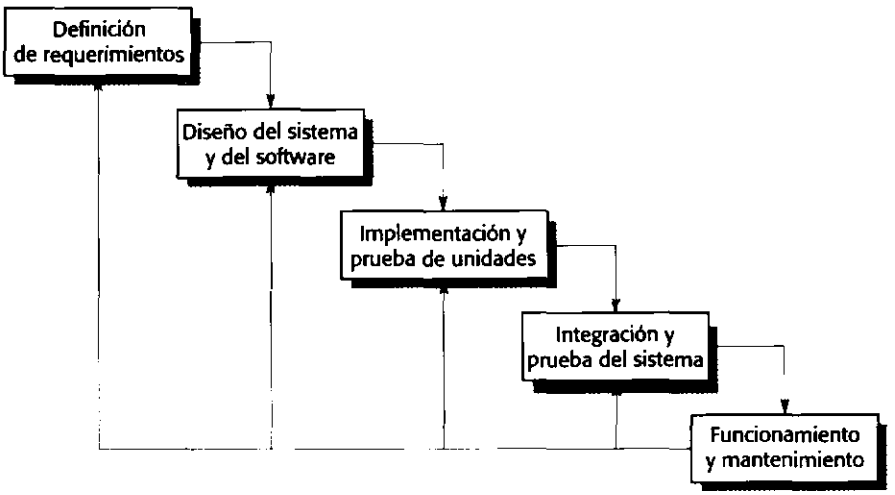


Figura 4.1 El ciclo de vida del software.

En principio, el resultado de cada fase es uno o más documentos aprobados («firmados»). La siguiente fase no debe empezar hasta que la fase previa haya finalizado. En la práctica, estas etapas se superponen y proporcionan información a las otras. Durante el diseño se identifican los problemas con los requerimientos; durante el diseño del código se encuentran problemas, y así sucesivamente. El proceso del software no es un modelo lineal simple, sino que implica una serie de iteraciones de las actividades de desarrollo.

Debido a los costos de producción y aprobación de documentos, las iteraciones son costosas e implican rehacer el trabajo. Por lo tanto, después de un número reducido de iteraciones, es normal congelar partes del desarrollo, como la especificación, y continuar con las siguientes etapas de desarrollo. Los problemas se posponen para su resolución, se pasan por alto o se programan. Este congelamiento prematuro de requerimientos puede implicar que el sistema no haga lo que los usuarios desean. También puede conducir a sistemas mal estructurados debido a que los problemas de diseño se resuelven mediante trucos de implementación.

Durante la fase final del ciclo de vida (funcionamiento y mantenimiento), el software se pone en funcionamiento. Se descubren errores y omisiones en los requerimientos originales del software. Los errores de programación y de diseño emergen y se identifica la necesidad de una nueva funcionalidad. Por tanto, el sistema debe evolucionar para mantenerse útil. Hacer estos cambios (mantenimiento del software) puede implicar repetir etapas previas del proceso.

Las ventajas del modelo en cascada son que la documentación se produce en cada fase y que éste cuadra con otros modelos del proceso de ingeniería. Su principal problema es su inflexibilidad al dividir el proyecto en distintas etapas. Se deben hacer compromisos en las etapas iniciales, lo que hace difícil responder a los cambios en los requerimientos del cliente.

Por lo tanto, el modelo en cascada sólo se debe utilizar cuando los requerimientos se comprendan bien y sea improbable que cambien radicalmente durante el desarrollo del sistema. Sin embargo, el modelo refleja el tipo de modelo de proceso usado en otros proyectos de la ingeniería. Por consiguiente, los procesos del software que se basan en este enfoque se siguen utilizando para el desarrollo de software, particularmente cuando éste es parte de proyectos grandes de ingeniería de sistemas.

4.1.2 Desarrollo evolutivo

El desarrollo evolutivo se basa en la idea de desarrollar una implementación inicial, exponiéndola a los comentarios del usuario y refinándola a través de las diferentes versiones hasta que se desarrolla un sistema adecuado (Figura 4.2). Las actividades de especificación,

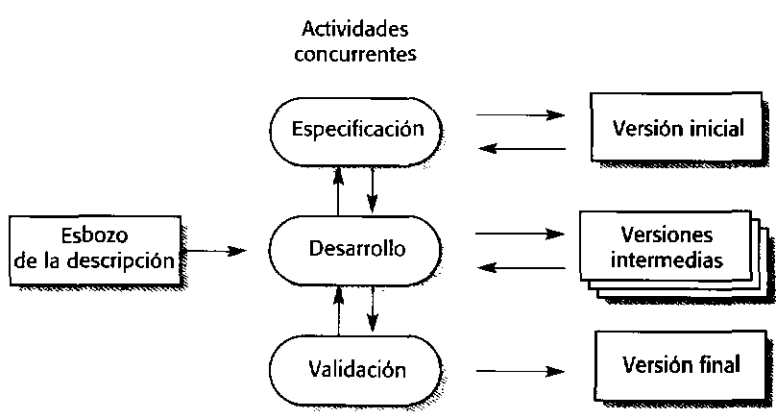


Figura 4.2
Desarrollo evolutivo.

desarrollo y validación se entrelazan en vez de separarse, con una rápida retroalimentación entre éstas.

Existen dos tipos de desarrollo evolutivo:

1. *Desarrollo exploratorio,* donde el objetivo del proceso es trabajar con el cliente para explorar sus requerimientos y entregar un sistema final. El desarrollo empieza con las partes del sistema que se comprenden mejor. El sistema evoluciona agregando nuevos atributos propuestos por el cliente.
2. *Prototipos desechables,* donde el objetivo del proceso de desarrollo evolutivo es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos para el sistema. El prototipo se centra en experimentar con los requerimientos del cliente que no se comprenden del todo.

En la producción de sistemas, un enfoque evolutivo para el desarrollo de software suele ser más efectivo que el enfoque en cascada, ya que satisface las necesidades inmediatas de los clientes. La ventaja de un proceso del software que se basa en un enfoque evolutivo es que la especificación se puede desarrollar de forma creciente. Tan pronto como los usuarios desarrollen un mejor entendimiento de su problema, éste se puede reflejar en el sistema software. Sin embargo, desde una perspectiva de ingeniería y de gestión, el enfoque evolutivo tiene dos problemas:

1. *El proceso no es visible.* Los administradores tienen que hacer entregas regulares para medir el progreso. Si los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada versión del sistema.
2. *A menudo los sistemas tienen una estructura deficiente.* Los cambios continuos tienden a corromper la estructura del software. Incorporar cambios en él se convierte cada vez más en una tarea difícil y costosa.

Para sistemas pequeños y de tamaño medio (hasta 500.000 líneas de código), el enfoque evolutivo de desarrollo es el mejor. Los problemas del desarrollo evolutivo se hacen particularmente agudos para sistemas grandes y complejos con un periodo de vida largo, donde diferentes equipos desarrollan distintas partes del sistema. Es difícil establecer una arquitectura del sistema estable usando este enfoque, el cual hace difícil integrar las contribuciones de los equipos.

Para sistemas grandes, se recomienda un proceso mixto que incorpore las mejores características del modelo en cascada y del desarrollo evolutivo. Esto puede implicar desarrollar un prototipo desechable utilizando un enfoque evolutivo para resolver incertidumbres en la especificación del sistema. Puede entonces reimplementarse utilizando un enfoque más estructurado. Las partes del sistema bien comprendidas se pueden especificar y desarrollar utilizando un proceso basado en el modelo en cascada. Las otras partes del sistema, como la interfaz de usuario, que son difíciles de especificar por adelantado, se deben desarrollar siempre utilizando un enfoque de programación exploratoria.

Los procesos del desarrollo evolutivo y el proceso de apoyo se estudian con más detalle en el Capítulo 17, junto con la construcción de prototipos de sistemas y el desarrollo rápido de aplicaciones. El desarrollo evolutivo también se incorpora en el Proceso Unificado de Rational que se trata más tarde en este capítulo.

4.1.3 Ingeniería del software basada en componentes

En la mayoría de los proyectos de software existe algo de reutilización de software. Por lo general, esto sucede informalmente cuando las personas que trabajan en el proyecto conocen di-

seños o código similares al requerido. Los buscan, los modifican según lo creen necesario y los incorporan en el sistema. En el enfoque evolutivo, descrito en la Sección 4.1.2, la reutilización es a menudo indispensable para el desarrollo rápido de sistemas.

Esta reutilización informal es independiente del proceso de desarrollo que se utilice. Sin embargo, en los últimos años, ha surgido un enfoque de desarrollo de software denominado ingeniería del software basada en componentes (CBSE) que se basa en la reutilización, el cual se está utilizando de forma amplia. Se introduce brevemente este enfoque aquí, pero se estudia con más detalle en el Capítulo 19.

Este enfoque basado en la reutilización se compone de una gran base de componentes software reutilizables y de algunos marcos de trabajo de integración para éstos. Algunas veces estos componentes son sistemas por sí mismos (COTS o sistemas comerciales) que se pueden utilizar para proporcionar una funcionalidad específica, como dar formato al texto o efectuar cálculos numéricos. En la Figura 4.3 se muestra el modelo del proceso genérico para la CBSE.

Aunque la etapa de especificación de requerimientos y la de validación son comparables con otros procesos, las etapas intermedias en el proceso orientado a la reutilización son diferentes. Estas etapas son:

1. *Análisis de componentes.* Dada la especificación de requerimientos, se buscan los componentes para implementar esta especificación. Por lo general, no existe una concordancia exacta y los componentes que se utilizan sólo proporcionan parte de la funcionalidad requerida.
2. *Modificación de requerimientos.* En esta etapa, los requerimientos se analizan utilizando información acerca de los componentes que se han descubierto. Entonces, estos componentes se modifican para reflejar los componentes disponibles. Si las modificaciones no son posibles, la actividad de análisis de componentes se puede llevar a cabo nuevamente para buscar soluciones alternativas.
3. *Diseño del sistema con reutilización.* En esta fase se diseña o se reutiliza un marco de trabajo para el sistema. Los diseñadores tienen en cuenta los componentes que se reutilizan y organizan el marco de trabajo para que los satisfaga. Si los componentes reutilizables no están disponibles, se puede tener que diseñar nuevo software.
4. *Desarrollo e integración.* Para crear el sistema, el software que no se puede adquirir externamente se desarrolla, y los componentes y los sistemas COTS se integran. En este modelo, la integración de sistemas es parte del proceso de desarrollo, más que una actividad separada.

La ingeniería del software basada en componentes tiene la ventaja obvia de reducir la cantidad de software a desarrollarse y así reduce los costos y los riesgos. Por lo general, también permite una entrega más rápida del software. Sin embargo, los compromisos en los requerimientos son inevitables, y esto puede dar lugar a un sistema que no cumpla las necesidades

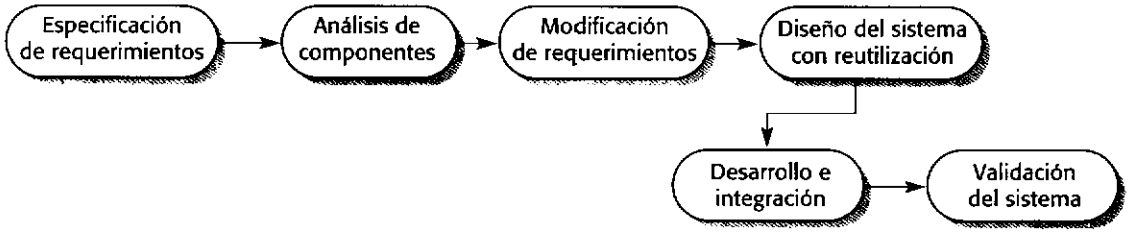


Figura 4.3 Ingeniería del software basada en componentes.

reales de los usuarios. Más aún: si las nuevas versiones de los componentes reutilizables no están bajo el control de la organización que los utiliza, se pierde parte del control sobre la evolución del sistema.

La CBSE tiene mucho en común con un enfoque que está surgiendo para el desarrollo de sistemas que se basa en la integración de servicios web de una serie de proveedores. En el Capítulo 12 se describe este enfoque de desarrollo de servicio céntrico.

4.2 Iteración de procesos

Los cambios son inevitables en todos los proyectos de software grandes. Los requerimientos del sistema cambian cuando el negocio que procura el sistema responde a las presiones externas. Las prioridades de gestión cambian. Cuando se dispone de nuevas tecnologías, cambian los diseños y la implementación. Esto significa que el proceso del software no es un proceso único; más bien, las actividades del proceso se repiten regularmente conforme el sistema se rehace en respuesta a peticiones de cambios.

El desarrollo iterativo es tan fundamental para el software que se le dedica un capítulo completo del libro más adelante (Capítulo 17). En esta sección, se introduce el tema describiendo dos modelos de procesos que han sido diseñados explícitamente para apoyar la iteración de procesos:

1. *Entrega incremental.* La especificación, el diseño y la implementación del software se dividen en una serie de incrementos, los cuales se desarrollan por turnos;
2. *Desarrollo en espiral.* El desarrollo del sistema gira en espiral hacia fuera, empezando con un esbozo inicial y terminando con el desarrollo final del mismo.

La esencia de los procesos iterativos es que la especificación se desarrolla junto con el software. Sin embargo, esto crea conflictos con el modelo de obtención de muchas organizaciones donde la especificación completa del sistema es parte del contrato de desarrollo del mismo. En el enfoque incremental, no existe una especificación completa del sistema hasta que el incremento final se especifica. Esto requiere un nuevo tipo de contrato, que a los clientes grandes como las agencias del gobierno les puede ser difícil de incorporar.

4.2.1 Entrega incremental

El modelo de desarrollo en cascada requiere que los clientes de un sistema cumplan un conjunto de requerimientos antes de que se inicie el diseño y que el diseñador cumpla estrategias particulares de diseño antes de la implementación. Los cambios de requerimientos implican rehacer el trabajo de captura de éstos, de diseño e implementación. Sin embargo, la separación en el diseño y la implementación deben dar lugar a sistemas bien documentados susceptibles de cambio. En contraste, un enfoque de desarrollo evolutivo permite que los requerimientos y las decisiones de diseño se retrasen, pero también origina un software que puede estar débilmente estructurado y difícil de comprender y mantener.

La entrega incremental (Figura 4.4) es un enfoque intermedio que combina las ventajas de estos modelos. En un proceso de desarrollo incremental, los clientes identifican, a grandes rasgos, los servicios que proporcionará el sistema. Identifican qué servicios son más importantes y cuáles menos. Entonces, se definen varios incrementos en donde cada uno proporciona un subconjunto de la funcionalidad del sistema. La asignación de servicios a los

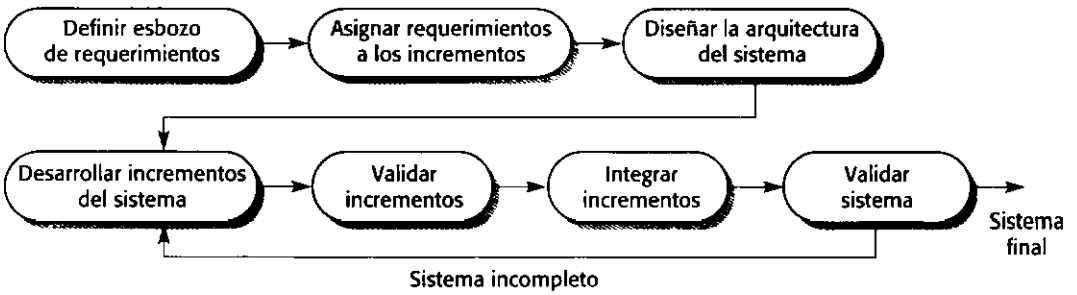


Figura 4.4 Entrega incremental.

incrementos depende de la prioridad del servicio con los servicios de prioridad más alta entregados primero.

Una vez que los incrementos del sistema se han identificado, los requerimientos para los servicios que se van a entregar en el primer incremento se definen en detalle, y éste se desarrolla. Durante el desarrollo, se puede llevar a cabo un análisis adicional de requerimientos para los requerimientos posteriores, pero no se aceptan cambios en los requerimientos para el incremento actual.

Una vez que un incremento se completa y entrega, los clientes pueden ponerlo en servicio. Esto significa que tienen una entrega temprana de parte de la funcionalidad del sistema. Pueden experimentar con el sistema, lo cual les ayuda a clarificar sus requerimientos para los incrementos posteriores y para las últimas versiones del incremento actual. Tan pronto como se completan los nuevos incrementos, se integran en los existentes de tal forma que la funcionalidad del sistema mejora con cada incremento entregado. Los servicios comunes se pueden implementar al inicio del proceso o de forma incremental tan pronto como sean requeridos por un incremento.

Este proceso de desarrollo incremental tiene varias ventajas:

1. Los clientes no tienen que esperar hasta que el sistema completo se entregue para sacar provecho de él. El primer incremento satisface los requerimientos más críticos de tal forma que pueden utilizar el software inmediatamente.
2. Los clientes pueden utilizar los incrementos iniciales como prototipos y obtener experiencia sobre los requerimientos de los incrementos posteriores del sistema.
3. Existe un bajo riesgo de un fallo total del proyecto. Aunque se pueden encontrar problemas en algunos incrementos, lo normal es que el sistema se entregue de forma satisfactoria al cliente.
4. Puesto que los servicios de más alta prioridad se entregan primero, y los incrementos posteriores se integran en ellos, es inevitable que los servicios más importantes del sistema sean a los que se les hagan más pruebas. Esto significa que es menos probable que los clientes encuentren fallos de funcionamiento del software en las partes más importantes del sistema.

Sin embargo, existen algunos problemas en el desarrollo incremental. Los incrementos deben ser relativamente pequeños (no más de 20.000 líneas de código) y cada uno debe entregar alguna funcionalidad del sistema. Puede ser difícil adaptar los requerimientos del cliente a incrementos de tamaño apropiado. Más aún, muchos de los sistemas requieren un conjunto de recursos que se utilizan en diferentes partes del sistema. Puesto que los requerimientos no se definen en detalle hasta que un incremento se implementa, puede ser difícil identificar los recursos comunes que requieren todos los incrementos.

Se ha desarrollado una variante de este enfoque incremental denominada *programación extrema* (Beck, 2000). Ésta se basa en el desarrollo y la entrega de incrementos de funcionalidad muy pequeños, en la participación del cliente en el proceso, en la mejora constante del código y en la programación por parejas. En el Capítulo 17 se estudia la programación por parejas y otros llamados métodos ágiles.

4.2.2 Desarrollo en espiral

El modelo en espiral del proceso del software (Figura 4.5) fue originalmente propuesto por Boehm (Boehm, 1988). Más que representar el proceso del software como una secuencia de actividades con retrospectiva de una actividad a otra, se representa como una espiral. Cada ciclo en la espiral representa una fase del proceso del software. Así, el ciclo más interno podría referirse a la viabilidad del sistema, el siguiente ciclo a la definición de requerimientos, el siguiente ciclo al diseño del sistema, y así sucesivamente.

Cada ciclo de la espiral se divide en cuatro sectores:

1. *Definición de objetivos*. Para esta fase del proyecto se definen los objetivos específicos. Se identifican las restricciones del proceso y el producto, y se traza un plan detallado de gestión. Se identifican los riesgos del proyecto. Dependiendo de estos riesgos, se planean estrategias alternativas.
2. *Evaluación y reducción de riesgos*. Se lleva a cabo un análisis detallado para cada uno de los riesgos del proyecto identificados. Se definen los pasos para reducir dichos riesgo. Por ejemplo, si existe el riesgo de tener requerimientos inapropiados, se puede desarrollar un prototipo del sistema.

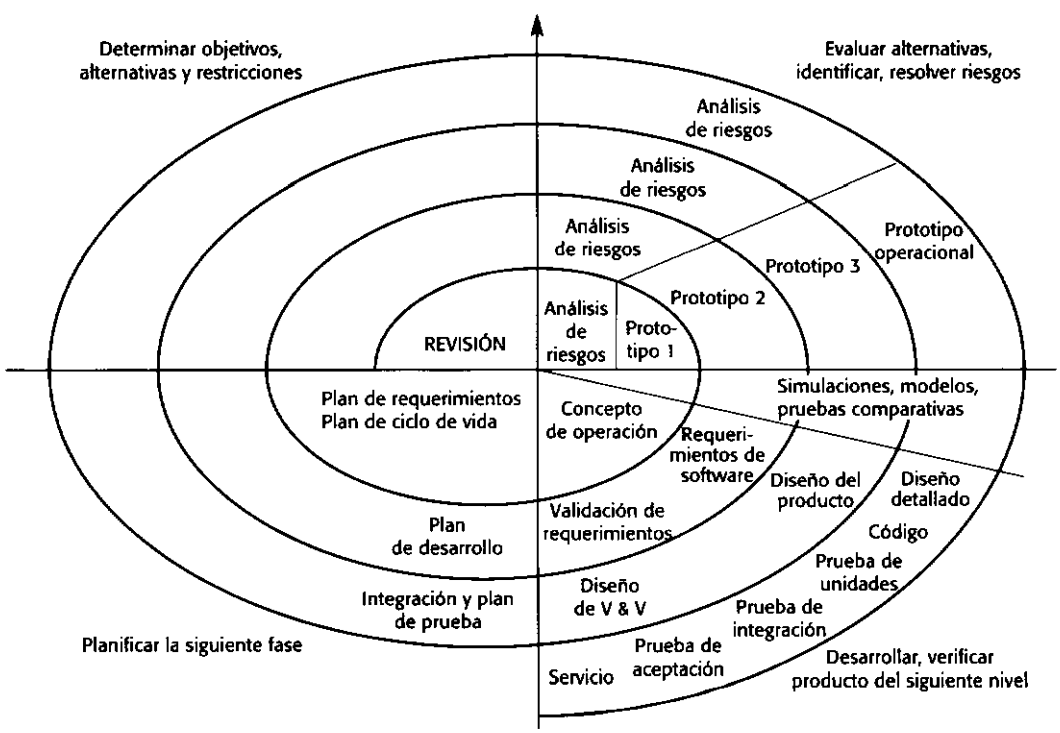


Figura 4.5 Modelo en espiral de Boehm para el proceso del software (©IEEE, 1988).

3. *Desarrollo y validación.* Después de la evaluación de riesgos, se elige un modelo para el desarrollo del sistema. Por ejemplo, si los riesgos en la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos. Si los riesgos de seguridad son la principal consideración, un desarrollo basado en transformaciones formales podría ser el más apropiado, y así sucesivamente. El modelo en cascada puede ser el más apropiado para el desarrollo si el mayor riesgo identificado es la integración de los subsistemas.
4. *Planificación.* El proyecto se revisa y se toma la decisión de si se debe continuar con un ciclo posterior de la espiral. Si se decide continuar, se desarrollan los planes para la siguiente fase del proyecto.

La diferencia principal entre el modelo en espiral y los otros modelos del proceso del software es la consideración explícita del riesgo en el modelo en espiral. Informalmente, el riesgo significa sencillamente algo que puede ir mal. Por ejemplo, si la intención es utilizar un nuevo lenguaje de programación, un riesgo es que los compiladores disponibles sean poco fiables o que no produzcan código objeto suficientemente eficiente. Los riesgos originan problemas en el proyecto, como los de confección de agendas y excesos en los costos; por lo tanto, la disminución de riesgos es una actividad muy importante en la gestión del proyecto. La gestión de los riesgos, una parte fundamental en la gestión de proyectos, se trata en el Capítulo 5.

Un ciclo de la espiral empieza con la elaboración de objetivos, como el rendimiento y la funcionalidad. Entonces se enumeran formas alternativas de alcanzar estos objetivos y las restricciones impuestas en cada una de ellas. Cada alternativa se evalúa contra cada objetivo y se identifican las fuentes de riesgo del proyecto. El siguiente paso es resolver estos riesgos mediante actividades de recopilación de información como la de detallar más el análisis, la construcción de prototipos y la simulación. Una vez que se han evaluado los riesgos, se lleva a cabo cierto desarrollo, seguido de una actividad de planificación para la siguiente fase del proceso.

4.3 Actividades del proceso

Las cuatro actividades básicas del proceso de especificación, desarrollo, validación y evolución se organizan de forma distinta en diferentes procesos del desarrollo. En el enfoque en cascada, están organizadas en secuencia, mientras que en el desarrollo evolutivo se entrelazan. Cómo se llevan a cabo estas actividades depende del tipo de software, de las personas y de la estructura organizacional implicadas. No hay una forma correcta o incorrecta de organizar estas actividades, y el objetivo de esta sección es simplemente proporcionar una introducción de cómo se pueden organizar.

4.3.1 Especificación del software

La especificación del software o ingeniería de requerimientos es el proceso de comprensión y definición de qué servicios se requieren del sistema y de identificación de las restricciones de funcionamiento y desarrollo del mismo. La ingeniería de requerimientos es una etapa particularmente crítica en el proceso del software ya que los errores en esta etapa originan inevitablemente problemas posteriores en el diseño e implementación del sistema.

En la Figura 4.6 se muestra el proceso de ingeniería de requerimientos. Éste conduce a la producción de un documento de requerimientos, que es la especificación del sistema. Normalmente en este documento los requerimientos se presentan en dos niveles de detalle. Los usuarios finales y los clientes necesitan una declaración de alto nivel de los requerimientos, mientras que los desarrolladores del sistema necesitan una especificación más detallada de éste.

Existen cuatro fases principales en el proceso de ingeniería de requerimientos:

1. *Estudio de viabilidad.* Se estima si las necesidades del usuario se pueden satisfacer con las tecnologías actuales de software y hardware. El estudio analiza si el sistema propuesto será rentable desde un punto de vista de negocios y si se puede desarrollar dentro de las restricciones de presupuesto existentes. Este estudio debe ser relativamente económico y rápido de elaborar. El resultado debe informar si se va a continuar con un análisis más detallado.
2. *Obtención y análisis de requerimientos.* Es el proceso de obtener los requerimientos del sistema por medio de la observación de los sistemas existentes, discusiones con los usuarios potenciales y proveedores, el análisis de tareas, etcétera. Esto puede implicar el desarrollo de uno o más modelos y prototipos del sistema que ayudan al analista a comprender el sistema a especificar.
3. *Especificación de requerimientos.* Es la actividad de traducir la información recopilada durante la actividad de análisis en un documento que define un conjunto de requerimientos. En este documento se pueden incluir dos tipos de requerimientos: los *requerimientos del usuario*, que son declaraciones abstractas de los requerimientos del cliente y del usuario final del sistema, y los *requerimientos del sistema*, que son una descripción más detallada de la funcionalidad a proporcionar.
4. *Validación de requerimientos.* Esta actividad comprueba la veracidad, consistencia y completitud de los requerimientos. Durante este proceso, inevitablemente se descubren errores en el documento de requerimientos. Se debe modificar entonces para corregir estos problemas.

Por supuesto, las actividades en el proceso de requerimientos no se llevan a cabo de forma estrictamente secuencial. El análisis de requerimientos continúa durante la definición y especificación, y a lo largo del proceso surgen nuevos requerimientos. Por lo tanto, las activida-

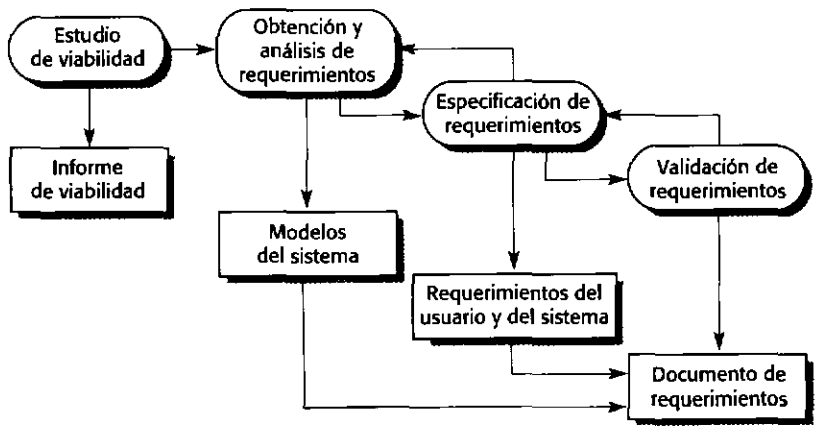


Figura 4.6 El proceso de la ingeniería de requerimientos.

des de análisis, definición y especificación se entrelazan. En los métodos ágiles como la programación extrema, los requerimientos de desarrollan de forma incremental conforme a las prioridades del usuario, y la obtención de requerimientos viene de los usuarios que forman parte del equipo de desarrollo.

4.3.2 Diseño e implementación del software

La etapa de implementación del desarrollo de software es el proceso de convertir una especificación del sistema en un sistema ejecutable. Siempre implica los procesos de diseño y programación de software, pero, si se utiliza un enfoque evolutivo de desarrollo, también puede implicar un refinamiento de la especificación del software.

Un diseño de software es una descripción de la estructura del software que se va a implementar, los datos que son parte del sistema, las interfaces entre los componentes del sistema y, algunas veces, los algoritmos utilizados. Los diseñadores no llegan inmediatamente a un diseño detallado, sino que lo desarrollan de manera iterativa a través de diversas versiones. El proceso de diseño conlleva agregar formalidad y detalle durante el desarrollo del diseño, y regresar a los diseños anteriores para corregirlos.

El proceso de diseño puede implicar el desarrollo de varios modelos del sistema con diferentes niveles de abstracción. Mientras se descompone un diseño, se descubren errores y omisiones de las etapas previas. Esta retroalimentación permite mejorar los modelos de diseño previos. La Figura 4.7 es un modelo de este proceso que muestra las descripciones de diseño que pueden producirse en varias etapas del diseño. Este diagrama sugiere que las etapas son secuenciales. En realidad, las actividades del proceso de diseño se entrelazan. La retroalimentación entre etapas y la consecuente repetición del trabajo es inevitable en todos los procesos de diseño.

Una especificación para la siguiente etapa es la salida de cada actividad de diseño. Esta especificación puede ser abstracta y formal, realizada para clarificar los requerimientos, o puede ser una especificación para determinar qué parte del sistema se va a construir. Durante todo el proceso de diseño, se detalla cada vez más esta especificación. El resultado final del proceso son especificaciones precisas de los algoritmos y estructuras de datos a implementarse.

Las actividades específicas del proceso de diseño son:

1. *Diseño arquitectónico*. Los subsistemas que forman el sistema y sus relaciones se identifican y documentan. En los Capítulos 11, 12 y 13 se trata este importante tema.

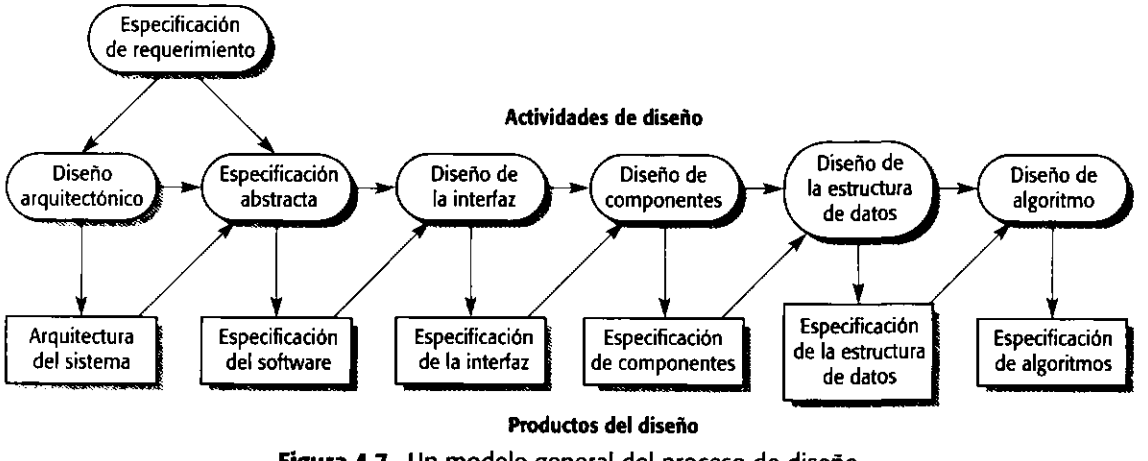


Figura 4.7 Un modelo general del proceso de diseño.

2. *Especificación abstracta*. Para cada subsistema se produce una especificación abstracta de sus servicios y las restricciones bajo las cuales debe funcionar.
3. *Diseño de la interfaz*. Para cada subsistema se diseña y documenta su interfaz con otros subsistemas. Esta especificación de la interfaz debe ser inequívoca ya que permite que el subsistema se utilice sin conocimiento de su funcionamiento. En esta etapa pueden utilizarse los métodos de especificación formal que se describen en el Capítulo 10.
4. *Diseño de componentes*. Se asignan servicios a los componentes y se diseñan sus interfaces.
5. *Diseño de la estructura de datos*. Se diseña en detalle y especifica la estructura de datos utilizada en la implementación del sistema.
6. *Diseño de algoritmos*. Se diseñan en detalle y especifican los algoritmos utilizados para proporcionar los servicios.

Éste es un modelo general del proceso de diseño, y los procesos reales y prácticos pueden adaptarlo de diversas maneras. He aquí algunas adaptaciones posibles:

1. Las dos últimas etapas —diseño de la estructura de datos y de algoritmos— se pueden retrasar hasta la etapa de implementación.
2. Si se utiliza un enfoque exploratorio de diseño, las interfaces del sistema se pueden diseñar después de que se especifiquen las estructuras de datos.
3. Se puede omitir la etapa de especificación abstracta, aunque normalmente es una parte fundamental del diseño de sistemas críticos.

Cada vez más, cuando se utilizan métodos ágiles de desarrollo (*véase* el Capítulo 17), las salidas del proceso de diseño no serán documentos de especificación separados, sino que estarán representadas en el código del programa. Una vez diseñada la arquitectura de un sistema, las etapas posteriores del diseño son incrementales. Cada incremento se representa como código del programa en vez de como un modelo de diseño.

Un enfoque opuesto es dado por los *métodos estructurados* que se basan en la producción de modelos gráficos del sistema (*véase* el Capítulo 8) y, en muchos casos, código automáticamente generado desde estos modelos. Los métodos estructurados se inventaron en los años 70 en apoyo del diseño orientado a funciones (Constantine y Yourdon, 1979; Gane y Sarson, 1979). Se propusieron varios modelos competentes de apoyo al diseño orientado a objetos (Robinson, 1992; Booch, 1994) y éstos se unificaron en los años 90 para crear el Lenguaje Unificado de Modelado (UML) y el proceso unificado de diseño asociado (Rumbaugh *et al.*, 1991; Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a; Rumbaugh *et al.*, 1999b). En el momento de escribir este libro, una revisión importante del UML (UML 2.0) está en marcha.

Un método estructurado incluye un modelo del proceso de diseño, notaciones para representar el diseño, formatos de informes, reglas y pautas de diseño. Los métodos estructurados pueden ayudar a alguno o a la totalidad de los siguientes modelos de un sistema:

1. Un modelo que objetos que muestra las clases de objetos utilizadas en el sistema y sus dependencias.
2. Un modelo de secuencias que muestra cómo interactúan los objetos en el sistema cuando éste se ejecuta.
3. Un modelo del estado de transición que muestra los estados del sistema y los disparadores de las transiciones desde un estado a otro.

4. Un modelo estructural en el cual se documentan los componentes del sistema y sus agregaciones.
5. Un modelo de flujo de datos en el que el sistema se modela utilizando la transformación de datos que tiene lugar cuando se procesan. Éste no se utiliza normalmente en los métodos orientados a objetos, pero todavía se utiliza frecuentemente en el diseño de sistemas de tiempo real y de negocio.

En la práctica, los «métodos» estructurados son realmente notaciones estándar que comprenden prácticas aceptables. Si se siguen estos métodos y se aplican las pautas, puede obtenerse un diseño razonable. La creatividad del diseñador aún se requiere para decidir la descomposición del sistema y asegurar que el diseño capte de forma adecuada la especificación del mismo. Los estudios empíricos de los diseñadores (Bansler y Bødker, 1993) muestran que éstos raramente siguen los métodos convencionales. Seleccionan y eligen de las pautas según circunstancias locales.

El desarrollo de un programa para implementar el sistema se sigue de forma natural del proceso de diseño. Aunque algunos programas, como los de los sistemas críticos de seguridad, se diseñan en detalle antes de que se inicie cualquier implementación, es más común que las primeras etapas de diseño y desarrollo de programas estén entrelazadas. Las herramientas CASE se pueden utilizar para generar un programa esqueleto a partir de un diseño. Esto incluye código para definir e implementar las interfaces, y en muchos casos el desarrollador sólo necesita agregar detalles del funcionamiento de cada componente del programa.

La programación es una actividad personal y no existe un proceso general que se siga comúnmente. Algunos programadores empiezan con los componentes que comprenden, los desarrollan y después continúan con los que comprenden menos. Otros toman el enfoque opuesto, dejando los componentes que son más familiares hasta el final debido a que saben cómo desarrollarlos. Algunos desarrolladores prefieren definir los datos al inicio del proceso y los utilizan para conducir el desarrollo del programa; otros dejan los datos sin especificar tanto como sea posible.

Normalmente, los programadores llevan a cabo algunas pruebas del código que han desarrollado. A menudo esto muestra defectos en el programa que se deben eliminar del mismo. Esto se denomina *depuración*. Las pruebas y la depuración de defectos son procesos diferentes. Las pruebas establecen la existencia de defectos. La depuración comprende la localización y corrección de estos defectos.

La Figura 4.8 ilustra las etapas de la depuración. Los defectos en el código se localizan y el programa se modifica para cumplir los requerimientos. Las pruebas se deben entonces repetir para asegurar que los cambios se han efectuado correctamente. Así, el proceso de depuración es parte tanto del desarrollo como de las pruebas del software.

Al depurar, se generan hipótesis acerca del comportamiento que se observa en el programa; después, se prueban estas hipótesis con la esperanza de encontrar el defecto que origina la anomalía en la salida. Probar las hipótesis puede implicar realizar una traza manual del código del programa. Pueden escribirse nuevos casos de pruebas para localizar el problema. Para ayudar al proceso de depuración, se pueden utilizar herramientas de depuración interactiva que muestran los valores intermedios de las variables del programa y una traza de las sentencias ejecutadas.

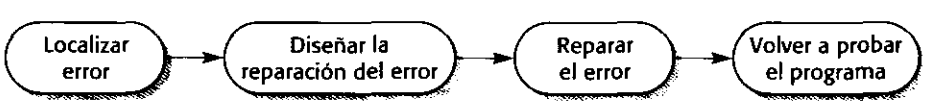


Figura 4.8 El proceso de depuración.

4.3.3 Validación del software

La validación del software o, de forma más general, la verificación y validación (V & V) se utiliza para mostrar que el sistema se ajusta a su especificación y que cumple las expectativas del usuario que lo comprará. Implica procesos de comprobación, como las inspecciones y revisiones (véase el Capítulo 22), en cada etapa del proceso del software desde la definición de requerimientos hasta el desarrollo del programa. Sin embargo, la mayoría de los costos de validación aparecen después de la implementación, cuando se prueba el funcionamiento del sistema (Capítulo 23).

A excepción de los programas pequeños, los sistemas no se deben probar como una simple unidad monolítica. La Figura 4.9 muestra un proceso de pruebas de tres etapas en el cual se prueban los componentes del sistema, la integración del sistema y, finalmente, el sistema con los datos del cliente. En el mejor de los casos, los defectos se descubren en las etapas iniciales del proceso y los problemas con la interfaz, cuando el sistema se integra. Sin embargo, cuando se descubren defectos el programa debe depurarse y esto puede requerir la repetición de otras etapas del proceso de pruebas. Los errores en los componentes del programa pueden descubrirse durante las pruebas del sistema. Por lo tanto, el proceso es iterativo y se retroalimenta tanto de las últimas etapas como de la primera parte del proceso.

Las etapas del proceso de pruebas son:

1. *Prueba de componentes (o unidades).* Se prueban los componentes individuales para asegurarse de que funcionan correctamente. Cada uno se prueba de forma independiente, sin los otros componentes del sistema. Los componentes pueden ser entidades simples como funciones o clases de objetos, o pueden ser agrupaciones coherentes de estas entidades.
2. *Prueba del sistema.* Los componentes se integran para formar el sistema. Este proceso comprende encontrar errores que son el resultado de interacciones no previstas entre los componentes y su interfaz. También comprende validar que el sistema cumpla sus requerimientos funcionales y no funcionales y probar las propiedades emergentes del sistema. Para sistemas grandes, esto puede ser un proceso gradual en el cual los componentes se integran para formar subsistemas que son probados individualmente antes de que ellos mismos se integren para formar el sistema final.
3. *Prueba de aceptación.* Es la etapa final en el proceso de pruebas antes de que se acepte que el sistema se ponga en funcionamiento. Éste se prueba con los datos proporcionados por el cliente más que con datos de prueba simulados. Debido a la diferencia existente entre los datos reales y los de prueba, la prueba de aceptación puede revelar errores y omisiones en la definición de requerimientos del sistema. También puede revelar problemas en los requerimientos donde los recursos del sistema no cumplen las necesidades del usuario o donde el desempeño del sistema es inaceptable.

Normalmente, el desarrollo de componentes y las pruebas se entrelazan. Los programadores definen sus propios datos de prueba y de forma incremental prueban el código que se

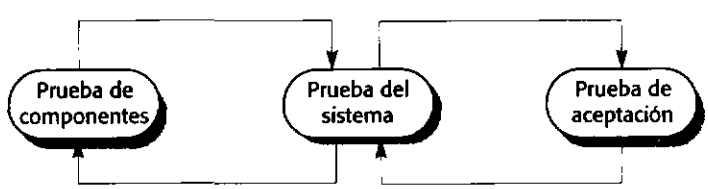


Figura 4.9 El proceso de pruebas.

va desarrollando. Éste es un enfoque económicamente razonable puesto que el programador es el que mejor conoce los componentes y es, por lo tanto, la mejor persona para generar los casos de prueba.

Si se utiliza un enfoque incremental de desarrollo, cada incremento debe ser probado cuando se desarrolla, con estas pruebas basadas en los requerimientos de ese incremento. En la programación extrema, las pruebas se desarrollan junto con los requerimientos antes de que empiece el desarrollo. Esto ayuda a los probadores y desarrolladores a entender los requerimientos y asegurar que no hay retardos pues se crean los casos de prueba.

Las últimas etapas de prueba consisten en integrar el trabajo de los programadores y deben planificarse por adelantado. Un equipo independiente de probadores debe trabajar a partir de planes de prueba que se desarrollan desde de la especificación y diseño del sistema. La Figura 4.10 ilustra cómo los planes de prueba son el vínculo entre las actividades de prueba y de desarrollo.

La prueba de aceptación algunas veces se denomina prueba alfa. Los sistemas personalizados se desarrollan para un único cliente. El proceso de prueba alfa continúa hasta que el desarrollador del sistema y el cliente acuerdan que el sistema que se va a entregar es una implementación aceptable de los requerimientos del sistema.

Cuando un sistema se va a comercializar como un producto de software, a menudo se utiliza un proceso de prueba denominado prueba beta. Ésta comprende la entrega de un sistema a un número potencial de clientes que acuerdan utilizarlo, los cuales informan de los problemas a los desarrolladores del sistema. Esto expone el producto a un uso real y detecta los errores no identificados por los constructores del sistema. Después de esta retroalimentación, el sistema se modifica y se entrega ya sea para una prueba beta adicional o para la venta.

4.3.4 Evolución del software

La flexibilidad de los sistemas software es una de las principales razones por la que más y más software se incorpora a los sistemas grandes y complejos. Una vez que se decide adquirir hardware, es muy costoso hacer cambios en su diseño. Sin embargo, se pueden hacer cambios al software en cualquier momento durante o después del desarrollo del sistema. Aun cambios importantes son todavía mucho más económicos que los correspondientes de los sistemas hardware.

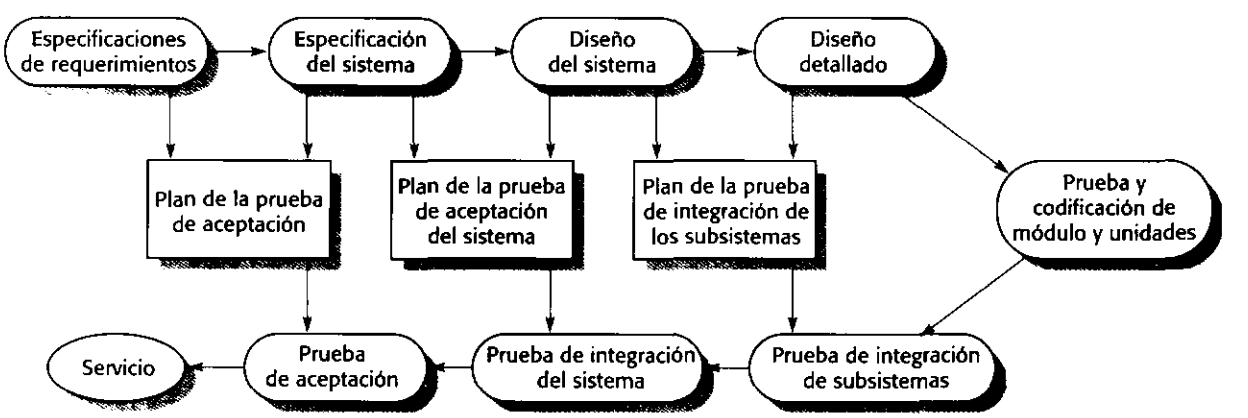


Figura 4.10 Las fases de prueba en el proceso del software.

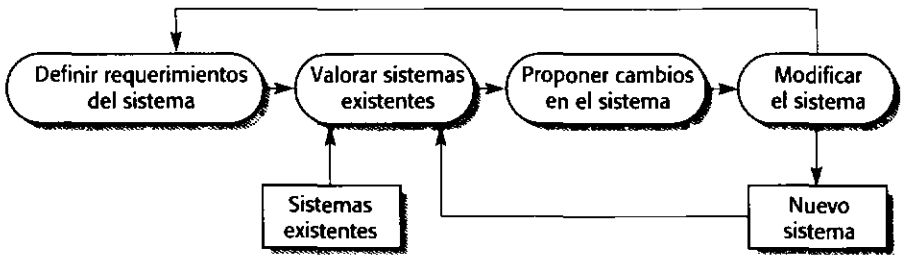


Figura 4.11 Evolución del sistema.

Históricamente, siempre ha existido una separación entre el proceso de desarrollo y el proceso de evolución del software (mantenimiento del software). La gente considera el desarrollo de software como una actividad creativa en la cual un sistema software se desarrolla desde un concepto inicial hasta que se pone en funcionamiento. Sin embargo, a veces consideran el mantenimiento del software como algo aburrido y sin interés. Aunque los costos de «mantenimiento» son a menudo varias veces los costos iniciales de desarrollo, el proceso de mantenimiento se considera a veces menos problemático que el desarrollo del software original.

Esta distinción entre el desarrollo y el mantenimiento es cada vez más irrelevante. Hoy en día, pocos sistemas software son completamente nuevos, lo que implica que tiene más sentido ver el desarrollo y el mantenimiento como actividades continuas. Más que dos procesos separados, es más realista considerar a la ingeniería del software como un proceso evolutivo (Figura 4.11) en el cual el software se cambia continuamente durante su periodo de vida como respuesta a los requerimientos cambiantes y necesidades del usuario.

4.4 El Proceso Unificado de Rational

El Proceso Unificado de Rational (RUP) es un ejemplo de un modelo de proceso moderno que proviene del trabajo en el UML y el asociado Proceso Unificado de Desarrollo de Software (Rumbaugh *et al.*, 1999b). Se ha incluido aquí una descripción ya que es un buen ejemplo de un modelo de proceso híbrido. Reúne elementos de todos los modelos de procesos genéricos (Sección 4.1), iteraciones de apoyo (Sección 4.2) e ilustra buenas prácticas en la especificación y el diseño (Sección 4.3).

El RUP reconoce que los modelos de procesos genéricos presentan un sola enfoque del proceso. En contraste, el RUP se describe normalmente desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo sobre el tiempo.
2. Una perspectiva estática que muestra las actividades del proceso que se representan.
3. Una perspectiva práctica que sugiere buenas prácticas a utilizar durante el proceso.

La mayor parte de las descripciones del RUP intentan combinar las perspectivas estática y dinámica es un único diagrama (Krutchen, 2000). Esto hace el proceso más difícil de entender, por lo que aquí se utilizan descripciones separadas de cada una de estas perspectivas.

El RUP es un modelo en fases que identifica cuatro fases diferentes en el proceso del software. Sin embargo, a diferencia del modelo en cascada donde las fases se equiparan con las actividades del proceso, las fases en el RUP están mucho más relacionadas con asuntos de negocio más que técnicos. La Figura 4.12 muestra las fases en el RUP. Éstas son:

1. *Inicio*. El objetivo de la fase de inicio es el de establecer un caso de negocio para el sistema. Se deben identificar todas las entidades externas (personas y sistemas) que

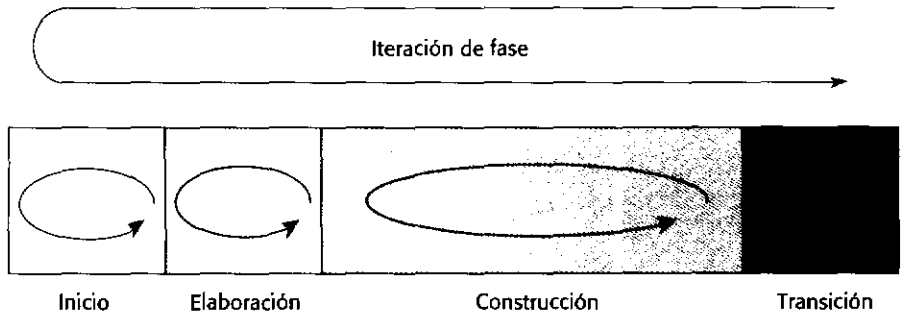


Figura 4.12 Fases del Proceso Unificado de Rational.

interactuarán con el sistema y definir estas interacciones. Esta información se utiliza entonces para evaluar la aportación que el sistema hace al negocio. Si esta aportación es de poca importancia, se puede cancelar el proyecto después de esta fase.

2. *Elaboración*. Los objetivos de la fase de elaboración son desarrollar una comprensión del dominio del problema, establecer un marco de trabajo arquitectónico para el sistema, desarrollar el plan del proyecto e identificar los riesgos clave del proyecto. Al terminar esta fase, se debe tener un modelo de los requerimientos del sistema (se especifican los casos de uso UML), una descripción arquitectónica y un plan de desarrollo del software.
3. *Construcción*. La fase de construcción fundamentalmente comprende el diseño del sistema, la programación y las pruebas. Durante esta fase se desarrollan e integran las partes del sistema. Al terminar esta fase, debe tener un sistema software operativo y la documentación correspondiente lista para entregarla a los usuarios.
4. *Transición*. La fase final del RUP se ocupa de mover el sistema desde la comunidad de desarrollo a la comunidad del usuario y hacerlo trabajar en un entorno real. Esto se deja de lado en la mayor parte de los modelos de procesos del software pero es, en realidad, una actividad de alto costo y a veces problemática. Al terminar esta fase, se debe tener un sistema software documentado que funciona correctamente en su entorno operativo.

La iteración dentro del RUP es apoyada de dos formas, como se muestra en la Figura 4.12. Cada fase se puede representar de un modo iterativo con los resultados desarrollados incrementalmente. Además, el conjunto entero de fases puede también representarse de forma incremental, como se muestra en la citada figura por la flecha en forma de bucle desde la Transición hasta el Inicio.

La vista estática del RUP se centra en las actividades que tienen lugar durante el proceso de desarrollo. Éstas se denominan *flujos de trabajo* en la descripción del RUP. Existen seis principales flujos de trabajo del proceso identificados en el proceso y tres principales flujos de trabajo de soporte. El RUP se ha diseñado conjuntamente con el UML —un lenguaje de modelado orientado a objetos—, por lo que la descripción del flujo de trabajo se orienta alrededor de los modelos UML asociados. En la Figura 4.13 se describen los principales flujos de trabajo de ingeniería y de soporte.

La ventaja de presentar perspectivas dinámicas y estáticas es que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. Al menos en principio, todos los flujos de trabajo del RUP pueden estar activos en todas las etapas del proceso. Por supuesto, la mayor parte del esfuerzo se realizará en flujos de trabajo tales como el modelado del negocio y los requerimientos en las primeras fases del proceso y en las pruebas y despliegue en las fases posteriores.

Modelado del negocio	**Los procesos del negocio se modelan utilizando casos de uso de negocio.**
Requerimientos	**Se definen los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema.**
Análisis y diseño	**Se crea y documenta un modelo del diseño utilizando modelos arquitectónicos, modelos de componentes, modelos de objetos y modelos de secuencias.**
Implementación	**Se implementan y estructuran en subsistemas los componentes del sistema. La generación automática de código de los modelos del diseño ayuda a acelerar este proceso.**
Pruebas	**Las pruebas son un proceso iterativo que se lleven a cabo conjuntamente con la implementación. A la finalización de la implementación tienen lugar las pruebas del sistema.**
Despliegue	**Se crea una release del producto, se distribuye a los usuarios y se instala en su lugar de trabajo.**
Configuración y cambios de gestión	**Este flujo de trabajo de soporte gestiona los cambios del sistema (véase el Cap. 29).**
Gestión del proyecto	**Este flujo de trabajo de soporte gestiona el desarrollo del sistema (véase el Cap. 5).**
Entorno	**Este flujo de trabajo se refiere a hacer herramientas software apropiadas disponibles para los equipos de desarrollo de software.**

Figura 4.13 Flujos de trabajo estáticos en el Proceso Unificado de Rational.

La perspectiva práctica en el RUP describe buenas prácticas de la ingeniería del software que son aconsejables en el desarrollo de sistemas. Se recomiendan seis buenas prácticas fundamentales:

1. *Desarrolle el software de forma iterativa.* Planifique incrementos del sistema basado en las prioridades del usuario y desarrollo y entregue las características del sistema de más alta prioridad al inicio del proceso de desarrollo.
2. *Gestione los requerimientos.* Documente explícitamente los requerimientos del cliente y manténgase al tanto de los cambios de estos requerimientos. Analice el impacto de los cambios en el sistema antes de aceptarlos.
3. *Utilice arquitecturas basadas en componentes.* Estructure la arquitectura del sistema en componentes como se indicó anteriormente en este capítulo.
4. *Modele el software visualmente.* Utilice modelos gráficos UML para presentar vistas estáticas y dinámicas del software.
5. *Verifique la calidad del software.* Asegure que el software cumple los estándares de calidad organizacionales.
6. *Controle los cambios del software.* Gestione los cambios del software usando un sistema de gestión de cambios y procedimientos y herramientas de gestión de configuraciones (véase el Capítulo 29).

El RUP no es un proceso apropiado para todos los tipos de desarrollo sino que representa una nueva generación de procesos genéricos. Las innovaciones más importantes son la separación de fases y los flujos de trabajo, y el reconocimiento de que la utilización del software en un entorno del usuario es parte del proceso. Las fases son dinámicas y tienen objetivos. Los flujos de trabajo son estáticos y son actividades técnicas que no están asociadas con fases únicas sino que pueden utilizarse durante el desarrollo para alcanzar los objetivos de cada fase.

4.5 Ingeniería del Software Asistida por computadora

Ingeniería del Software Asistida por Computadora (CASE) es el nombre que se le da al software que se utiliza para ayudar a las actividades del proceso del software como la ingeniería de requerimientos, el diseño, el desarrollo de programas y las pruebas. Por lo tanto, las herramientas CASE incluyen editores de diseño, diccionarios de datos, compiladores, depuradores, herramientas de construcción de sistemas, etcétera.

La tecnología CASE proporciona ayuda al proceso del software automatizando algunas de sus actividades, así como proporcionando información acerca del software en desarrollo. Algunos ejemplos de las actividades que se pueden automatizar utilizando CASE son:

1. El desarrollo de modelos gráficos del sistema como parte de la especificación de requerimientos o del diseño de software.
2. La comprensión del diseño utilizando un diccionario de datos que tiene información sobre las entidades y relaciones del diseño.
3. La generación de interfaces de usuario a partir de la descripción gráfica de la interfaz que es elaborada de forma interactiva por el usuario.
4. La depuración de programas por medio de la provisión de la información proporcionada por los programas en ejecución.
5. La conversión automática de programas de una versión anterior de una lenguaje de programación, como COBOL, a una versión más reciente.

La tecnología CASE está disponible para la mayoría de las actividades rutinarias en el proceso del software. Esto permite algunas mejoras en la calidad y productividad del software, aunque éstas sean menores que las predichas por los primeros partidarios de CASE. Éstos sugirieron que se tendría una mejora mayor si se utilizaran entornos CASE integrados. En realidad, las mejoras reales son del 40% (Huff, 1992). Aunque esto es significante, las predicciones que se hicieron cuando se introdujeron las herramientas CASE en los años 80 y 90 fueron que el uso de la tecnología CASE generaría enormes ahorros en los costos del proceso del software.

Las mejoras por la utilización de CASE están limitadas por dos factores:

1. Esencialmente, la ingeniería del software es una actividad de diseño que se basa en la creatividad. Los sistemas CASE automatizan las actividades rutinarias, pero los intentos de utilizar la inteligencia artificial para proporcionar ayuda al diseño no han tenido éxito.
2. En la mayoría de las organizaciones, la ingeniería del software es una actividad de equipo, y los ingenieros invierten mucho tiempo interactuando con los otros miembros del equipo. La tecnología CASE no proporciona mucha ayuda para esto.

Actualmente, la tecnología CASE está madura y hay herramientas disponibles y bancos de trabajo de un amplio rango de proveedores. Sin embargo, más que centrarse en alguna herramienta específica, aquí se presenta una visión general, con algunos comentarios de apoyo específico en otros capítulos. En la página web se incluyen enlaces a otro material de CASE y a proveedores de herramientas CASE.

4.5.1 Clasificación de CASE

Las clasificaciones de CASE nos ayudan a comprender los tipos de herramientas CASE y su papel en la ayuda a las actividades de proceso del software. Existen varias formas diferentes

de clasificar las herramientas CASE, cada una de las cuales nos proporciona una perspectiva distinta de estas herramientas. En esta sección, se describen dichas herramientas desde tres de estas perspectivas:

1. *Una perspectiva funcional* en la que las herramientas CASE se clasifican de acuerdo con su función específica.
2. *Una perspectiva de proceso* en la que las herramientas se clasifican de acuerdo con las actividades del proceso que ayudan.
3. *Una perspectiva de integración* en la que las herramientas CASE se clasifican de acuerdo con la forma en que están organizadas en unidades integradas que proporcionan ayuda a una o más actividades del proceso.

La Figura 4.14 es una clasificación de las herramientas CASE acorde con su función. Esta tabla enumera diferentes tipos de herramientas CASE y da ejemplos específicos de cada una. Ésta no es una lista completa de herramientas CASE. Las herramientas especializadas, como las de ayuda a la reutilización, no se incluyen.

La Figura 4.15 presenta una clasificación alternativa de las herramientas CASE. Muestra las fases del proceso que reciben ayuda por varios tipos de herramientas CASE. Las herramientas para la planificación y estimación, edición de texto, preparación de documentos y gestión de la configuración pueden utilizarse durante todo el proceso del software.

Ora dimensión de clasificación posible es la amplia ayuda que ofrece la tecnología CASE para el proceso del software. Fuggetta (Fuggetta, 1993) propone que los sistemas CASE se deben clasificar en tres categorías:

1. *Las herramientas ayudan* a las tareas individuales del proceso como la verificación de la consistencia de un diseño, la compilación de un programa y la comparación de los resultados de las pruebas. Las herramientas pueden ser de propósito general, independientes (por ejemplo, un procesador de texto) o agrupadas en bancos de trabajo.
2. *Los bancos de trabajo ayudan* a las fases o actividades del proceso como la especificación, el diseño, etcétera. Normalmente consisten en un conjunto de herramientas con algún grado mayor o menor de integración.

Herramientas de planificación	Herramientas PERT, herramientas de estimación, hojas de cálculo.
Herramientas de edición	Editores de texto, editores de diagramas, procesadores de texto.
Herramientas de gestión del cambio	Herramientas de rastreo de requerimientos, sistemas de control de cambios.
Herramientas de gestión de la configuración	Sistema de gestión de las versiones, herramientas de construcción de sistemas.
Herramientas de construcción de prototipos	Lenguajes de muy alto nivel, generadores de interfaz de usuario.
Herramientas de apoyo a métodos	Editores de diseño, diccionarios de datos, generadores de código.
Herramientas de procesamiento de lenguajes	Compiladores, intérpretes.
Herramientas de análisis de programas	Generadores de referencias cruzadas, analizadores estáticos, analizadores dinámicos.
Herramientas de pruebas	Generadores de pruebas de datos, comparadores de archivos.
Herramientas de depuración	Sistemas de depuración interactiva.
Herramientas de documentación	Programas de diseño de páginas, editores de imágenes.
Herramientas de reingeniería	Sistemas de referencias cruzadas, sistemas reestructuración de programas.

Figura 4.14 Clasificación funcional de las herramientas CASE.

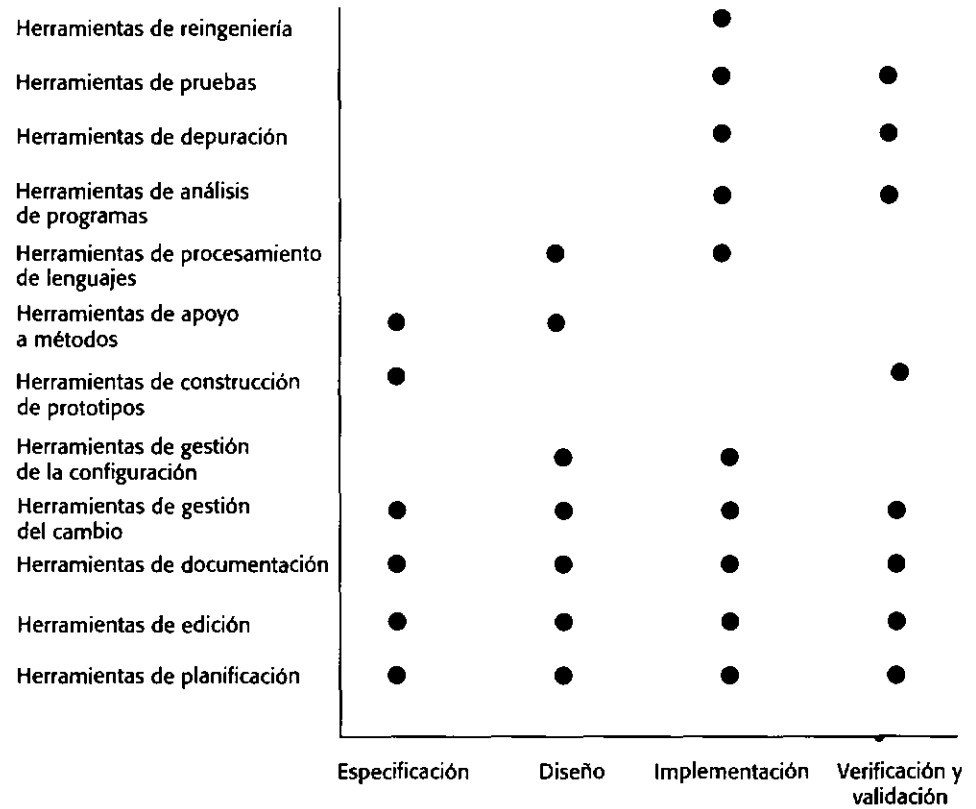


Figura 4.15 Clasificación basada en actividades de las herramientas CASE.

3. *Los entornos ayudan* a todos los procesos del software, o al menos a una parte sustancial de éstos. Normalmente incluyen varios bancos de trabajo integrados.

La Figura 4.16 ilustra esta clasificación y muestra algunos ejemplos de estas clases de ayuda CASE. Por supuesto, esto es un ejemplo ilustrativo; muchos tipos de herramientas y bancos de trabajo se han quedado fuera de este diagrama.

Las herramientas de propósito general se utilizan a discreción del ingeniero de software quien toma decisiones acerca de cuándo aplicarlas para ayudar al proceso. Sin embargo, los bancos de trabajo por lo general ayudan a algún método que incluye un modelo del proceso y un conjunto de reglas/pautas que se aplican al software en desarrollo. Los entornos se clasifican en integrados y centrados en el proceso. Los entornos integrados proporcionan ayuda a los datos, al control y a la integración de la presentación. Los entornos centrados en procesos son más generales. Incluyen el conocimiento del proceso del software y un motor de procesos que utiliza este modelo del proceso para aconsejar a los ingenieros sobre qué herramientas o bancos de trabajo hay que aplicar y cuándo deben utilizarse.

En la práctica, los límites entre estas diferentes clases son borrosos. Las herramientas se pueden vender como productos individuales, pero pueden proporcionar ayuda a diferentes actividades. Por ejemplo, la mayoría de los procesadores de texto incluyen un editor de diagramas integrado. Los bancos de trabajo CASE para el diseño normalmente ayudan a la programación y a las pruebas, de tal forma que se relacionan más con el entorno que con los bancos de trabajo especializados. Por tanto, puede que no siempre resulte fácil ubicar un producto utilizando una clasificación. No obstante, la clasificación proporciona un primer paso útil para ayudar a entender la amplitud del soporte que una herramienta proporciona al proceso.

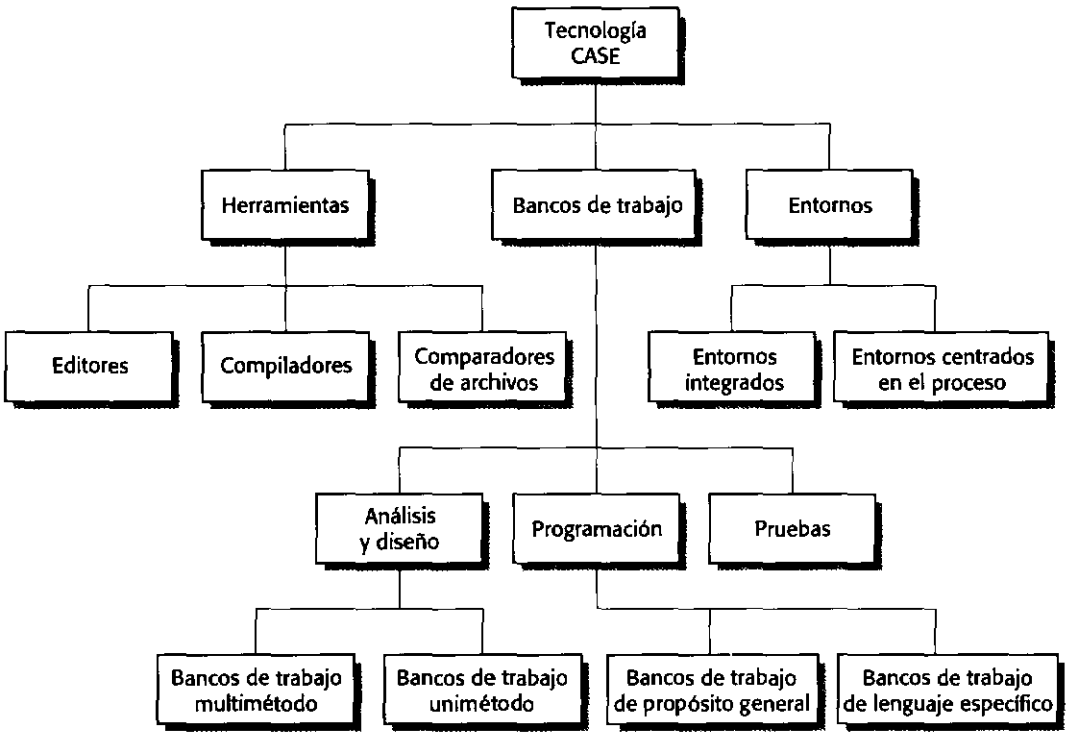


Figura 4.16 Herramientas, bancos de trabajo y entornos.

PUNTOS CLAVE

- Los procesos del software son las actividades relacionadas con la producción de un sistema software. Los modelos del proceso del software son representaciones abstractas de estos procesos.
- Todos los procesos del software incluyen la especificación, el diseño, la implementación, la validación y la evolución del software.
- Los modelos genéricos del proceso describen la organización de los procesos del software. Ejemplos de estos modelos son el modelo en cascada, el desarrollo evolutivo y la ingeniería del software basada en componentes.
- Los modelos de iteración de procesos presentan el proceso del software como un ciclo de actividades. La ventaja de este enfoque es que evita compromisos prematuros con una especificación o diseño. Ejemplos de este tipo de modelos son el desarrollo incremental y el modelo en espiral.
- La ingeniería de requerimientos es el proceso de desarrollar una especificación del software. Las especificaciones pretenden comunicar las necesidades del sistema del cliente a los desarrolladores del sistema.
- Los procesos de diseño e implementación comprenden la transformación de la especificación de los requerimientos en un sistema software ejecutable. Los métodos sistemáticos de diseño se pueden utilizar como parte de esta transformación.

- **La validación del software es el proceso de verificar que el sistema se ajusta a su especificación y que satisface las necesidades reales de los usuarios del sistema.**
- **La evolución del software se interesa en modificar los sistemas software existentes para cumplir los nuevos requerimientos. Esto se está convirtiendo en el enfoque normal para el desarrollo de sistemas de pequeño y mediano tamaño.**
- **El Proceso Unificado de Rational es un modelo del proceso moderno y genérico que se organiza en fases (inicio, elaboración, construcción y transición), pero que separa las actividades (requerimientos, análisis y diseño, etc.) de estas fases.**
- **La tecnología CASE proporciona ayuda automatizada a los procesos del software. Las herramientas CASE ayudan a las actividades individuales del proceso; los bancos de trabajo ayudan a un conjunto de actividades relacionadas; los entornos ayudan a todas o a la mayoría de las actividades del proceso del software.**

LECTURAS ADICIONALES

Extreme Programming Explained: Embrace Change. Un libro evangélico que describe el proceso y las experiencias de la programación extrema. El autor fue el inventor de la programación extrema y comunica muy bien su entusiasmo. (Kent Beck, 2000, Addison-Wesley.)

The Rational Unified Process – An Intruduction. Éste era el trabajo más interesante disponible sobre RUP cuando se estaba escribiendo este libro. Krutchen describe bien el proceso, pero se echa en falta una exposición más detallada de las dificultades prácticas que presenta su uso. (P. Krutchen, 2000, Addison-Wesley.)

Managing Software Quality and Business Risk. Éste es principalmente un libro sobre la gestión del software, pero incluye un capítulo excelente (Capítulo 4) sobre los modelos de proceso. (M. Ould, 1999, John Wiley & Sons.)

«A classification of CASE technology». El esquema de clasificación propuesto en este artículo se ha utilizado en este capítulo, pero Fuggetta entra en más detalle e ilustra cómo varios productos comerciales encajan en este esquema. [A. Fuggetta, *IEEE Computer,* 26 (12), diciembre de 1993.]

EJERCICIOS

4.1 Sugiera el modelo de proceso del software genérico que podría utilizarse para gestionar el desarrollo de los siguientes sistemas, dando algunas razones basadas en el tipo de sistema a desarrollar:

- Un sistema de control antibloqueo de frenos de un automóvil.
- Un sistema de realidad virtual para ayudar al mantenimiento del software.
- Un sistema de contabilidad universitaria que reemplace el existente.
- Un sistema interactivo que permita a los pasajeros encontrar los horarios de los trenes a partir de las terminales instaladas en las estaciones.

4.2 Explique por qué los programas que se desarrollan utilizando el desarrollo evolutivo tienden a ser difíciles de mantener.

4.3 Explique cómo el modelo en cascada para el proceso del software y el de construcción de prototipos pueden encajar en el de proceso en espiral.

4.4 ¿Cuáles son las ventajas de proporcionar vistas estáticas y dinámicas del proceso del software como en el Proceso Unificado de Rational?

4.5 Explique por qué es importante hacer distinción entre el desarrollo de los requerimientos del usuario y el de los requerimientos del sistema en el proceso de ingeniería de requerimientos.

4.6 Describa las principales actividades en el proceso de diseño del software y las salidas de estas actividades. Utilizando un diagrama, muestre las posibles relaciones entre las salidas.

4.7 ¿Cuáles son los cinco componentes de un método de diseño? Considere cualquier método que conozca y describa sus componentes. Evalúe la integridad del método elegido.

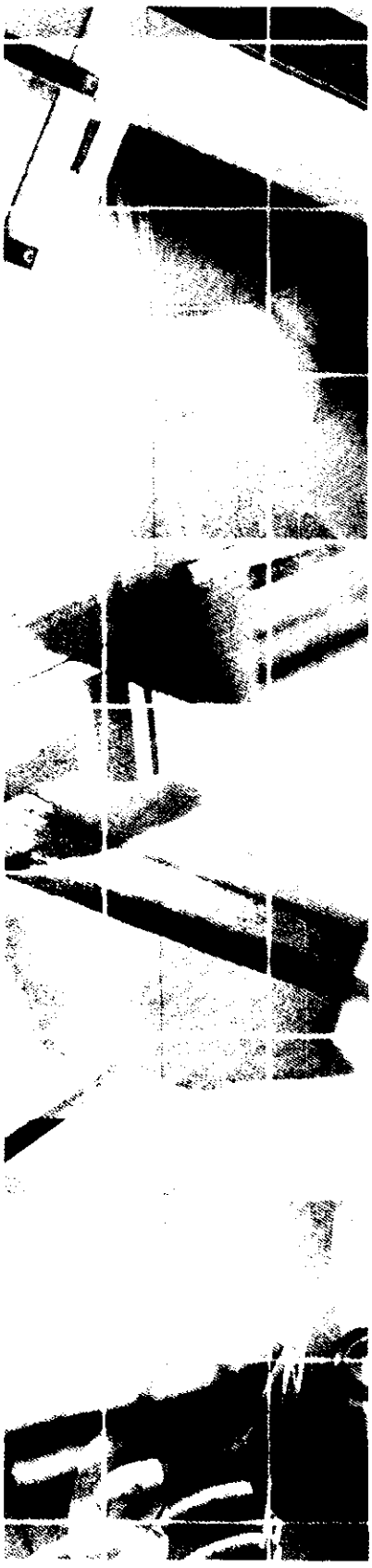
4.8 Diseñe un modelo de proceso para las pruebas de ejecución y recopile los resultados.

4.9 Explique por qué un sistema software que se utiliza en un entorno real debe cambiar o convertirse progresivamente en menos útil.

4.10 Indique cómo el esquema de clasificación de la tecnología CASE puede ser útil para los administradores encargados de adquirir sistemas CASE.

4.11 Haga un estudio de las herramientas disponibles en su entorno local de desarrollo y clasifíquelas de acuerdo con los parámetros (función, actividad, amplitud de soporte) sugeridos aquí.

4.12 Históricamente, la introducción de la tecnología ha causado profundos cambios en el mercado laboral y, al menos temporalmente, elimina personas de los puestos de trabajo. Comente si es probable que la introducción de tecnología CASE avanzada pueda tener las mismas consecuencias para los ingenieros de software. Si piensa que no es así, explique por qué no. Si piensa que reducirá las oportunidades de trabajo, ¿es ético para los ingenieros afectados resistirse, pasivamente o activamente, a la introducción de esta tecnología?



5 Gestión de proyectos

Objetivos

El objetivo de este capítulo es dar un panorama de la gestión de proyectos de software. Cuando termine de leer este capítulo:

- conocerá las tareas principales de los gestores de proyectos de software.
- comprenderá por qué la naturaleza del software hace más difícil la gestión de proyectos software que la gestión de los proyectos de otras ingenierías.
- comprenderá por qué planificar proyectos es esencial en todos los proyectos de software.
- conocerá la forma en que las representaciones gráficas (gráficos de barras y redes de actividades) son utilizadas por los gestores de proyectos para representar las agendas del proyecto.
- conocerá el proceso de gestión de riesgos y algunos de los riesgos que surgen en los proyectos de software.

Contenidos

La gestión de proyectos de software es una parte esencial de la ingeniería del software. La buena gestión no puede garantizar el éxito del proyecto. Sin embargo, la mala gestión usualmente lleva al fracaso del proyecto. El software es entregado tarde, los costes son mayores que los estimados y los requerimientos no se cumplen.

Los gestores de software son responsables de la planificación y temporalización del desarrollo de los proyectos. Supervisan el trabajo para asegurar que se lleva a cabo conforme a los estándares requeridos y supervisan el progreso para comprobar que el desarrollo se ajusta al tiempo previsto y al presupuesto. La administración de proyectos de software es necesaria debido a que la ingeniería de software profesional siempre está sujeta a restricciones organizacionales de tiempo y presupuesto. El trabajo del gestor de proyectos de software es asegurar que éstos cumplan dichas restricciones y entregar software que contribuya a las metas de la compañía de desarrollo software.

Los gestores de software hacen el mismo tipo de trabajo que otros gestores. Sin embargo, la ingeniería del software es diferente en varios aspectos a otros tipos, lo que hace a la gestión de software particularmente difícil. Algunas de estas diferencias son las siguientes:

1. *El producto es intangible.* El gestor de un proyecto de construcción de un embarcadero o de uno de ingeniería civil puede ver el producto mientras se está desarrollando. Si hay un desfase en calendario, el efecto en el producto es visible de forma obvia: partes de la estructura no están completas. El software es intangible. No se puede ver ni tocar. Los gestores de proyectos de software no pueden ver el progreso. Confían en otros para elaborar la documentación necesaria para revisar el progreso.
2. *No existen procesos del software estándar.* En las disciplinas de ingeniería con larga historia, el proceso se prueba y verifica. Para tipos particulares de sistemas, como puentes o edificios, el proceso de ingeniería se comprende bien. Sin embargo, los procesos de software varían notablemente de una organización a otra. A pesar de que la comprensión del proceso del software se ha desarrollado de forma significativa en los últimos años, aún no se puede predecir con certeza cuándo un proceso particular tiende a desarrollar problemas. Esto es especialmente cierto cuando el proyecto software es parte un proyecto de ingeniería de un sistema grande.
3. *A menudo los proyectos grandes son únicos.* Por lo general, los proyectos grandes de software son diferentes de proyectos previos. En consecuencia, los gestores, aun cuando cuenten con una amplia experiencia, ésta no es suficiente para anticipar los problemas. Más aún, los rápidos cambios tecnológicos en las computadoras y las comunicaciones hacen parecer obsoleta la experiencia previa. Las lecciones aprendidas en esas experiencias pueden no ser transferibles a los nuevos proyectos.

Debido a estos problemas, no es sorprendente que algunos proyectos de software se retrasen, sobrepasen el presupuesto y se entreguen fuera de tiempo. A menudo, los sistemas de software son nuevos y tecnológicamente innovadores. Frecuentemente los proyectos de ingeniería innovadores (como los nuevos sistemas de transporte) también tienen problemas de temporalización. Dadas las mezclas de dificultades, es notable que muchos proyectos de software sean entregados a tiempo y según lo presupuestado.

La gestión de proyectos de software es un tema amplio y no puede tratarse en un solo capítulo. Por lo tanto, en este capítulo se introduce el tema y se describen tres actividades importantes de gestión: planificación, calendarización de proyectos y gestión de riesgos. Los últimos capítulos (en la Parte 6) tratan otros aspectos de la gestión de software, entre los que se incluyen la gestión de personal, la estimación de los costes de software y la gestión de la calidad.

5.1 Actividades de gestión

Es imposible redactar una descripción estándar del trabajo de un gestor de software. El trabajo difiere enormemente dependiendo de la organización y del producto de software a desarrollar. Sin embargo, en algún momento, muchos gestores son responsables de algunas o de la totalidad de las siguientes actividades:

- Redacción de la propuesta
- Planificación y calendarización del proyecto
- Estimación de costes del proyecto
- Supervisión y revisión del proyecto
- Selección y evaluación del personal
- Redacción y presentación de informes

La primera etapa de un proyecto de software implica redactar una propuesta para realizar ese proyecto. La propuesta describe los objetivos del proyecto y cómo se llevaría a cabo. Por lo general, incluye estimaciones de coste y tiempo y justifica por qué el contrato del proyecto se le debe dar a una organización o a un equipo en particular. La redacción de la propuesta es una tarea crítica, ya que la existencia de muchas organizaciones de software depende de las propuestas aceptadas y los contratos asignados. No existen guías para esta tarea; la redacción de propuestas es una habilidad que se adquiere con la práctica y la experiencia.

La planificación de proyectos se refiere a la identificación de actividades, hitos y entregas de un proyecto. Por lo tanto, se debe bosquejar un plan para guiar el desarrollo hacia las metas del proyecto. La estimación del coste es una actividad relacionada con la estimación de los recursos requeridos para llevar a cabo el plan del proyecto. Este aspecto se tratará con mayor detalle más adelante en este capítulo y en el Capítulo 26.

La supervisión de proyecto es una actividad continua. El gestor debe tener conocimiento del progreso del proyecto y comparar el progreso con los costes actuales y los planificados. Aunque muchas organizaciones tienen mecanismos formales para supervisar, un gestor hábil podría formarse una imagen clara de lo que pasa llevando a cabo una entrevista informal con el personal del proyecto.

La supervisión informal predice problemas importantes del proyecto, y revela dificultades que pueden aparecer. Por ejemplo, las entrevistas diarias con el personal del proyecto pueden exteriorizar un problema en un fallo del software. Más que esperar un informe de atraso del proyecto, el gestor de software podría asignar algún experto para resolver el problema o podría decidir si se vuelve a programar.

Durante un proyecto, es normal tener varias revisiones formales de su gestión. Se hace la revisión completa del progreso y de los desarrollos técnicos del proyecto, y se tiene en cuenta el estado del proyecto junto con los propósitos de la organización que ha encargado el software.

El resultado de una revisión puede dar lugar a la cancelación del proyecto. El tiempo de desarrollo para un proyecto grande de software puede ser de varios años. Durante ese tiempo los objetivos organizacionales tienden obviamente a cambiar. Estos cambios pueden significar que el software ya no se necesita o que los requerimientos originales del proyecto son inapropiados. La gestión puede decidir parar el desarrollo del software o cambiar el proyecto para adecuarlo a los cambios de los objetivos de la organización.

Por lo general, los gestores de proyectos tienen que seleccionar a las personas para trabajar en su proyecto. De forma ideal, habrá personal disponible con habilidades y experiencia

apropiada para trabajar en el proyecto. Sin embargo, en muchos casos, los gestores tienen que establecer un equipo ideal mínimo para el proyecto. Las razones que explican esto son:

1. El presupuesto del proyecto no cubre la contratación de personal con sueldos altos. Se tiene que contratar personal con menos experiencia y menor sueldo.
2. El personal con experiencia apropiada no está disponible dentro o fuera de la organización. Es imposible reclutar nuevo personal para el proyecto. Dentro de la organización, los mejores trabajadores ya se han asignado a otros proyectos.
3. La organización desea desarrollar las habilidades de sus empleados. El personal inexperto puede ser asignado al proyecto para aprender y adquirir experiencia.

El gestor de software tiene que trabajar con estas restricciones al seleccionar al personal del proyecto. Sin embargo, todos estos problemas son probables a menos que exista un miembro del proyecto que cuente con algo de experiencia en el tipo de sistema a desarrollar. Sin esta experiencia, probablemente se cometerán muchos errores pequeños. En el Capítulo 25 se abordará la formación del equipo de desarrollo y la selección del personal.

Los gestores del proyecto son responsables de informar a los clientes y contratistas sobre el proyecto. Tienen que redactar documentos concisos y coherentes que resuman la información crítica de los informes detallados del proyecto. Les debe ser posible presentar esta información durante las revisiones de progreso. En consecuencia, comunicarse efectivamente de forma oral y escrita es una habilidad esencial que un gestor de proyectos debe tener.

5.2 Planificación del proyecto

La gestión efectiva de un proyecto de software depende de planificar completamente el progreso del proyecto. El gestor del proyecto debe anticiparse a los problemas que puedan surgir, así como preparar soluciones a esos problemas. Un plan, preparado al inicio de un proyecto, debe utilizarse como un conductor para el proyecto. Este plan inicial debe ser el mejor posible de acuerdo con la información disponible. Éste evolucionará conforme el proyecto progrese y la información sea mejor.

En la Sección 5.2.1 se describe una estructura para un plan de desarrollo de software. Además de un plan del proyecto, los gestores tienen que preparar otros tipos de planes, los cuales se describen brevemente en la Figura 5.1 y serán tratados con más detalle en otros capítulos del libro.

Plan de calidad	**Describe los procedimientos y los estándares de calidad que se utilizarán en un proyecto. Véase el Capítulo 24.**
Plan de validación	**Describe el enfoque, los recursos y la programación utilizados para la validación del sistema.**
Plan de gestión de configuraciones	**Describe los procedimientos para la gestión de configuraciones y las estructuras a utilizar. Véase el Capítulo 29.**
Plan de mantenimiento	**Predice los requerimientos del mantenimiento del sistema, los costes del mantenimiento y el esfuerzo requerido. Véase el Capítulo 27.**
Plan de desarrollo del personal	**Describe cómo se desarrollan las habilidades y experiencia de los miembros del equipo del proyecto.**

Figura 5.1 Tipos de plan.

```
Establecer las restricciones del proyecto
Hacer la valoración inicial de los parámetros del proyecto
Definir los hitos del proyecto y productos a entregar
Mientras el proyecto no se haya completado o cancelado repetir
        Bosquejar la programación en el tiempo del proyecto
        Iniciar actividades acordes con la programación
        Esperar (por un momento)
        Revisar el progreso del proyecto
        Revisar las estimaciones de los parámetros del proyecto
        Actualizar la programación del proyecto
        Renegociar las restricciones del proyecto y los productos a entregar
        Si (surgen problemas) entonces
            Iniciar la revisión técnica y la posible revisión
        fin de si
fin de repetir
```

Figura 5.2
Planificación del proyecto.

El pseudocódigo que se muestra en la Figura 5.2 describe el proceso de planificación del proyecto para el desarrollo de software. Muestra que la planificación es un proceso iterativo que solamente se completa cuando el proyecto mismo se termina. Conforme la información se hace disponible, el plan debe revisarse regularmente. Las metas globales del negocio son un factor importante que debe considerarse cuando se formula el plan del proyecto. Conforme éstas cambien, serán necesarios cambios en el proyecto.

El proceso de planificación se inicia con una valoración de las restricciones que afectan al proyecto (fecha de entrega requerida, personal disponible, presupuesto global, etcétera). Ésta se lleva a cabo en conjunto con una estimación de los parámetros del proyecto, como su estructura, tamaño y distribución de funciones. Entonces se definen los hitos de progreso y productos a entregar. En ese momento, el proceso entra en un ciclo. Se prepara un calendario para el proyecto y las actividades definidas en el calendario se inician o se continúan. Después de algún tiempo (por lo general 2 o 3 semanas), se revisa el proyecto y se señalan las discrepancias. Debido a que las estimaciones iniciales de los parámetros del proyecto son aproximaciones, el plan siempre deberá actualizarse.

Cuando se dispone de más información, los gestores del proyecto revisan las suposiciones del proyecto y la agenda. Si el proyecto se retrasa, tienen que renegociar con el cliente las restricciones del mismo y las entregas. Si esta renegociación no tiene éxito y no se puede cumplir el calendario, se debe llevar a cabo una revisión técnica. El objetivo de esta revisión es encontrar un enfoque alternativo que se ajuste a las restricciones del proyecto y cumpla con las metas del calendario.

Por supuesto, los gestores de proyectos inteligentes no suponen que todo irá bien. Durante el proyecto siempre surgen problemas en algunas descripciones. Las suposiciones iniciales y el calendario deben ser más bien pesimistas que optimistas. Debe haber suficiente holgura para que las contingencias en el plan, las restricciones del proyecto y los hitos no se tengan que negociar cada vez que se efectúa un ciclo en el plan.

5.2.1 El plan del proyecto

El plan del proyecto fija los recursos disponibles, divide el trabajo y crea un calendario de trabajo. En algunas organizaciones, el plan del proyecto es un único documento que incluye todos los diferentes tipos de planes (Figura 5.1). En otros casos, este plan sólo se refiere al proceso de desarrollo. Otros pueden estar referenciados, pero son proporcionados por separado.

El plan que se describe aquí tiene que ver con el último tipo de plan mencionado. Los detalles de este plan varían dependiendo del tipo de proyecto y de la organización. Sin embargo, muchos planes incluyen las siguientes secciones:

1. *Introducción*. Describe brevemente los objetivos del proyecto y expone las restricciones (por ejemplo, presupuesto, tiempo, etcétera) que afectan a la gestión del proyecto.
2. *Organización del proyecto*. Describe la forma en que el equipo de desarrollo está organizado, la gente involucrada y sus roles en el equipo.
3. *Análisis de riesgo*. Describe los posibles riesgos del proyecto, la probabilidad de que surjan estos riesgos y las estrategias de reducción de riesgos propuestas. La gestión de riesgos se estudia en la Sección 5.4.
4. *Requerimientos de recursos de hardware y software*. Describe el hardware y el software de ayuda requeridos para llevar a cabo el desarrollo. Si es necesario comprar hardware, se deben incluir las estimaciones de los precios y las fechas de entrega.
5. *División del trabajo*. Describe la división del proyecto en actividades e identifica los hitos y productos a entregar asociados con cada actividad. En la Sección 5.2.2 se indican los hitos y productos a entregar.
6. *Programa del proyecto*. Describe las dependencias entre actividades, el tiempo estimado requerido para alcanzar cada hito y la asignación de la gente a las actividades.
7. *Mecanismos de supervisión e informe*. Describe la gestión de informes y cuándo deben producirse, así como los mecanismos de supervisión del proyecto a utilizar.

El plan del proyecto debe revisarse regularmente durante el proyecto. Algunas partes, como el calendario del proyecto, cambiarán frecuentemente; otras serán más estables. Para simplificar las revisiones, se debe organizar el documento en secciones separadas que permitan su reemplazo de forma individual conforme evoluciona el plan.

5.2.2 Hitos y entregas

Los gestores necesitan información para hacer su trabajo. Como el software es intangible, esta información sólo se puede proveer como documentos que describan el estado del software que se está desarrollando. Sin esta información, es imposible juzgar el progreso y no se pueden actualizar los costes y calendarios.

Cuando se planifica un proyecto, se debe establecer una serie de *hitos* —puntos finales de una actividad del proceso del software—. En cada uno, debe existir una salida formal, como un informe, que se debe presentar al gestor. Los informes de hitos no deben ser documentos amplios. Deben ser informes cortos de los logros en una actividad del proyecto. Los hitos deben representar el fin de una etapa lógica en el proyecto. Los hitos indefinidos como «80 % del código completo» son imposibles de validar y carecen de utilidad para la gestión del proyecto. No podemos validar si se ha llegado a esta etapa debido a que la cantidad de código que se tiene que desarrollar no es precisa.

Una entrega es el resultado del proyecto que se entrega al cliente. De forma general, se entrega al final de una fase principal del proyecto como la especificación, el diseño, etcétera. Como regla general, las entregas son hitos, pero éstos no son necesariamente entregas. Dichos hitos pueden ser resultados internos del proyecto que son utilizados por el gestor del proyecto para verificar el progreso del mismo pero que no se entregan al cliente.

Para establecer los hitos, el proceso del software debe dividirse en actividades básicas con sus salidas asociadas. Por ejemplo, la Figura 5.3 muestra las actividades involucradas en la especificación de requerimientos cuando se utiliza la construcción de prototipos para ayudar

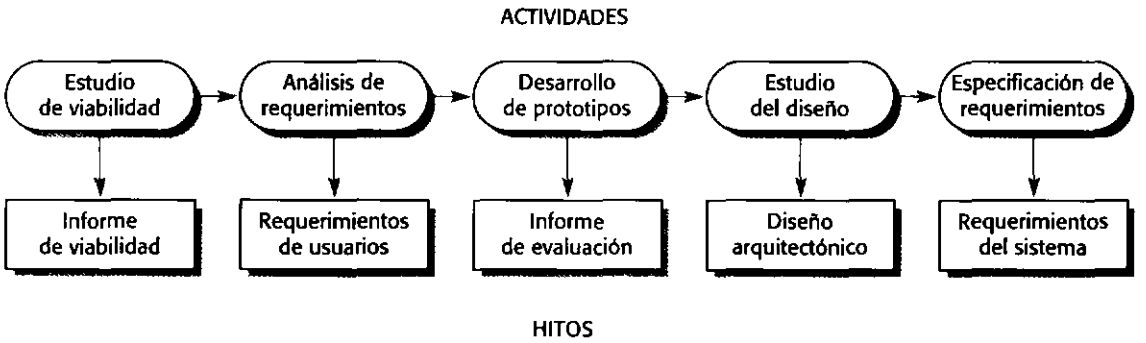


Figura 5.3 Hitos del proceso de especificación de requerimientos.

a validar dichos requerimientos. También se muestran las salidas principales de cada actividad (los hitos del proyecto). Las entregas del proyecto, las cuales son entregadas al cliente, son la definición y especificación de requerimientos.

5.3 Calendarización del proyecto

Ésta es una de las tareas más difíciles para los gestores de proyectos. Los gestores estiman el tiempo y los recursos requeridos para completar las actividades y organizarlas en una sucesión coherente. A menos que el proyecto a calendarizar sea similar a otro anterior, las estimaciones previas son una base incierta para la calendarización del nuevo proyecto. La estimación del calendario se complica más por el hecho de que proyectos diferentes pueden utilizar métodos de diseño y lenguajes de implementación diferentes.

Si el proyecto es técnicamente complejo, las estimaciones iniciales casi siempre son optimistas aun cuando los gestores traten de considerar las eventualidades. A este respecto, la calendarización del tiempo para la creación del software no es diferente a la de cualquier otro tipo de proyecto grande y complejo. Los nuevos aeroplanos, los puentes e incluso los nuevos modelos de automóviles se retrasan debido a problemas no anticipados. Por lo tanto, los calendarios se deben actualizar continuamente en la medida que se disponga de mejor información acerca del progreso.

La calendarización del proyecto (véase la Figura 5.4) implica separar todo el trabajo de un proyecto en actividades complementarias y considerar el tiempo requerido para completar dichas actividades. Por lo general, algunas de éstas se llevan a cabo en paralelo. Debemos coordinar estas actividades paralelas y organizar el trabajo para que la mano de obra se utilice

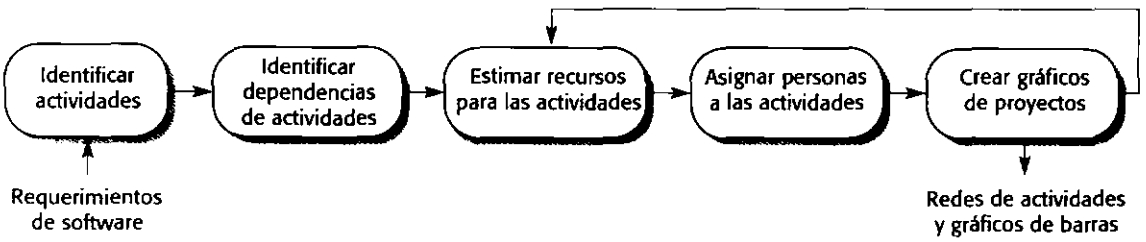


Figura 5.4 El proceso de calendarización del proyecto.

de forma óptima. Deben evitarse situaciones en que el proyecto entero se retrase debido a que no se ha terminado una actividad crítica.

Normalmente, las actividades del proyecto deben durar por lo menos una semana. Hacer subdivisiones más finas significa invertir una cantidad desproporcionada de tiempo en la estimación y revisión de tablas. También es útil asignar una cantidad de tiempo máxima de 8 a 10 semanas para realizar cualquier actividad. Si lleva más tiempo, se deben hacer subdivisiones.

Al estimar la calendarización, los gestores no deben suponer que cada etapa del proyecto estará libre de problemas. Las personas que trabajan en él pueden enfermarse o renunciar, el hardware puede fallar y el software o hardware de soporte puede llegar tarde. Si el proyecto es nuevo y técnicamente complejo, ciertas partes podrían ser más complicadas y llevarían más tiempo del que se estimó originalmente.

Como en los calendarios, los gestores deben estimar los recursos necesarios para completar cada tarea. El recurso principal es el esfuerzo humano que se requiere. Otros recursos pueden ser el espacio en disco requerido en un servidor, el tiempo requerido de hardware especializado, un simulador o el presupuesto para viajes del personal del proyecto. En el Capítulo 26 se trata con mayor detalle este tema.

Una buena regla práctica es estimar como si nada fuera a salir mal, y entonces incrementar la estimación para abarcar los problemas previstos. Con este mismo fin, a la estimación se le debe agregar un factor de contingencia adicional. Este factor extra de contingencia depende del tipo de proyecto, de los parámetros del proceso (fecha de entrega, estándares, etcétera) y de la calidad y experiencia de los ingenieros de software que trabajen en el proyecto. Como regla, para los problemas previstos siempre debe agregarse un 30 % a la estimación original y otro 20 % para cubrir algunas cosas no previstas.

Por lo general, el calendario del proyecto se representa como un conjunto de gráficos que muestran la división del trabajo, las dependencias de las actividades y la asignación del personal. Esto se aborda en la siguiente sección. Por lo general, las herramientas de gestión de software, como Microsoft Project, se utilizan para automatizar la producción de diagramas.

5.3.1 Gráficos de barras y redes de actividades

Los gráficos de barras y las redes de actividades son notaciones gráficas que se utilizan para ilustrar la calendarización del proyecto. Los gráficos de barras muestran quién es responsable de cada actividad y cuándo debe comenzar y finalizar ésta. Las redes de actividades muestran las dependencias entre las diferentes actividades que conforman un proyecto. Los gráficos de barras y las redes de actividades se generan automáticamente a partir de una base de datos de la información del proyecto utilizando una herramienta de gestión de proyectos.

Considere el conjunto de actividades que se muestra en la Figura 5.5. Esta tabla recoge las tareas, la duración e interdependencias de éstas. En ella se observa que la Tarea T3 depende de la Tarea T1. Esto significa que T1 debe completarse antes de que comience T3. Por ejemplo, T1 podría ser la preparación de un diseño de un componente y T3 la implementación de ese diseño. Antes de que se inicie la implementación, el diseño debe estar terminado.

Dadas la dependencia y la duración estimada de las actividades, una red de actividades muestra la sucesión de actividades que deben generarse (véase la Figura 5.6). Ésta muestra qué actividades se llevan a cabo en paralelo y cuáles deben ejecutarse en secuencia debido a la dependencia con una actividad previa. Las actividades se representan como rectángulos; los

T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Figura 5.5 Duración y dependencias de las tareas.

hitos y las entregas se muestran con esquinas redondeadas. Las fechas en este gráfico muestran la fecha de inicio de la actividad (están escritas en estilo británico, en el cual el día precede al mes). La red se debe leer de izquierda a derecha y de arriba abajo.

En la herramienta de gestión de proyecto utilizada para elaborar este gráfico, todas las actividades deben terminar en hitos. Una comienza cuando su hito precedente (puede depender de varias actividades) se ha alcanzado. Por lo tanto, en la tercera columna la Figura 5.5 se muestra el hito correspondiente (por ejemplo, M5) que se alcanza cuando las tareas en esa columna terminan (véase la Figura 5.6).

Antes de que se pueda pasar de un hito a otro, todas las trayectorias deben completarse. Por ejemplo, la tarea T9, que se muestra en la Figura 5.6, no se puede iniciar hasta que las tareas T3 y T6 se terminen.

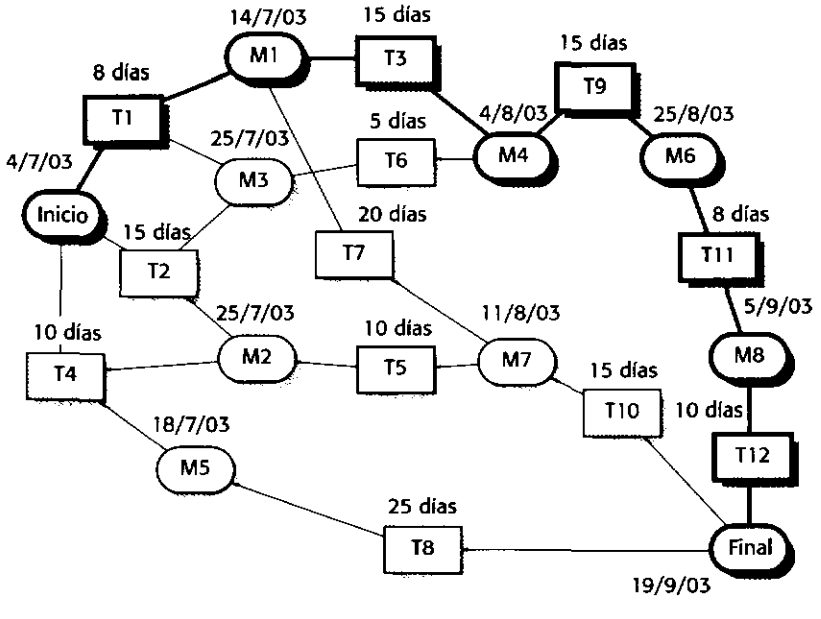


Figura 5.6 Una red de actividades.

El tiempo mínimo requerido para finalizar el proyecto se estima teniendo en cuenta la trayectoria más larga en la red de actividades (el camino crítico). En este caso, es 11 semanas de tiempo transcurrido o 55 días laborales. En la Figura 5.6 el camino crítico se muestra como una sucesión de rectángulos más resaltados. El calendario completo del proyecto depende de dicho camino. Cualquier aplazamiento al completar una actividad crítica provoca retraso en el proyecto, ya que las actividades siguientes no pueden comenzar hasta que se finalice la actividad retrasada.

Sin embargo, los retrasos de las actividades que no están ligadas al camino crítico no provocan necesariamente un aplazamiento en todo el calendario. Mientras los retrasos no se propaguen a estas actividades como para que se exceda el tiempo total del camino crítico, el proyecto no se verá afectado. Por ejemplo, si T8 se retrasa, esto no afectará a la fecha final de terminación del proyecto puesto que no se encuentra sobre el camino crítico. El gráfico de barras (véase la Figura 5.7) del proyecto muestra la holgura de los posibles retrasos como barras sombreadas.

Los gestores también utilizan las redes de actividades para asignar los trabajos en el proyecto. Dichas redes pueden dar una percepción de la dependencia de las actividades que de forma intuitiva no son obvias. Es posible modificar el diseño del sistema de tal forma que se acorte al camino crítico. El calendario del proyecto se puede acortar debido a que se reduce la cantidad de tiempo de espera para finalizar las actividades.

Inevitablemente, las agendas del proyecto iniciales serán incorrectas. A medida que el proyecto se desarrolla, las estimaciones deben ser comparadas con los tiempos reales. Esta comparación puede ser utilizada como base para una revisión de la agenda de las siguientes partes del proyecto. Cuando se muestran los tiempos reales, debemos revisar el gráfico de

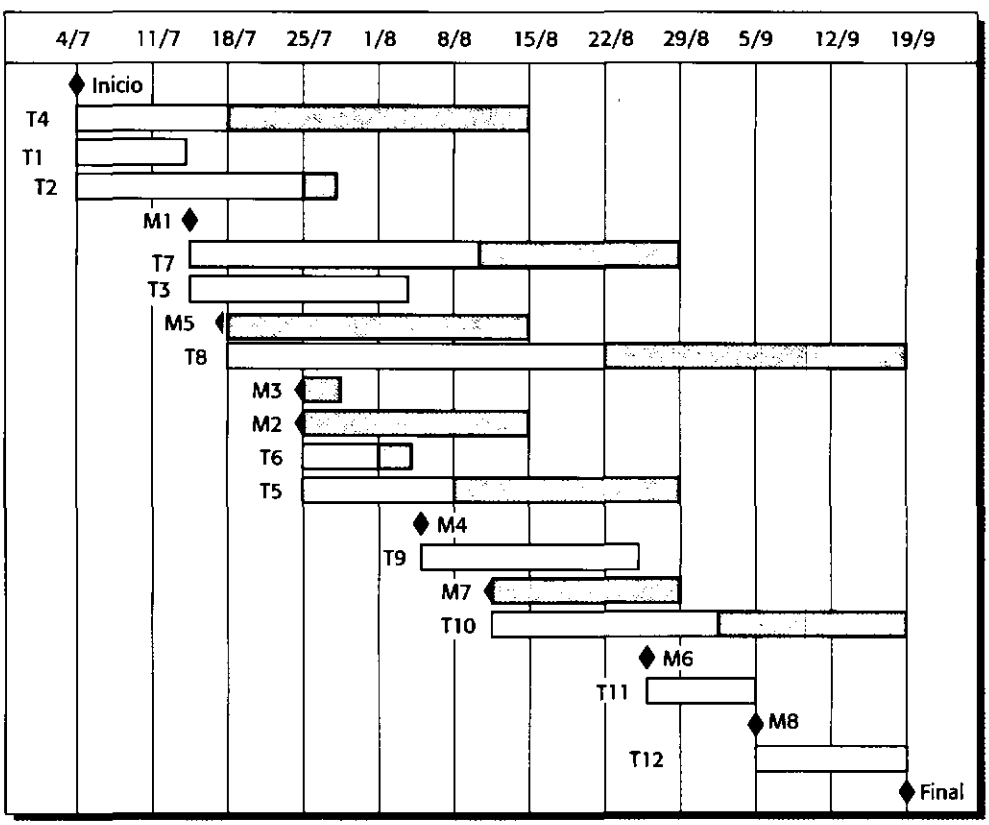


Figura 5.7 Gráfico de barras de actividades.

actividades. Las actividades siguientes del proyecto pueden ser reorganizadas para reducir la duración del camino crítico.

La Figura 5.7 es una forma alternativa de representar la información del calendario del proyecto. Es un gráfico de barras (algunas veces llamado gráfico de Gantt, su inventor) que muestra el calendario de un proyecto y las fechas iniciales y finales de las actividades. Algunas de las actividades que se muestran en la Figura 5.7 son seguidas por una barra sombreada cuya longitud es calculada por una herramienta de calendarización. Esto muestra que existe alguna flexibilidad en las fechas de terminación de estas actividades. Si alguna de éstas no se completa a tiempo, el camino crítico no se verá afectado hasta el final del periodo marcado por la barra sombreada. Las actividades que caen sobre dicho camino no tienen margen de error y se distinguen por no tener asociada una barra sombreada.

Además de considerar la calendarización, los gestores de proyectos también deben tener en cuenta la asignación de recursos y, en particular, la asignación de personal a las actividades del proyecto. Esta asignación puede ser introducida también desde las herramientas de gestión de proyectos, y los gráficos de barras generados muestran el personal del proyecto (Figura 5.8). Las personas no tienen por qué estar asignadas al proyecto en todo momento. Durante diferentes periodos pueden estar de vacaciones, trabajando en otros proyectos, en cursos de formación o realizando otras actividades.

Por lo general, las grandes organizaciones emplean varios especialistas para que trabajen en el proyecto cuando sea necesario. En la Figura 5.8 se puede ver que Mary y Jim son especialistas que sólo trabajan en una actividad del proyecto. Esto puede provocar problemas en la calendarización. Si un proyecto se retrasa mientras un especialista trabaja en él, puede causar efectos contundentes en los otros proyectos. Estos proyectos serán retrasados ya que el especialista no estará disponible.

5.4 Gestión de riesgos

Una tarea importante del gestor de proyectos es anticipar los riesgos que podrían afectar a la programación del proyecto o a la calidad del software a desarrollar y emprender acciones para

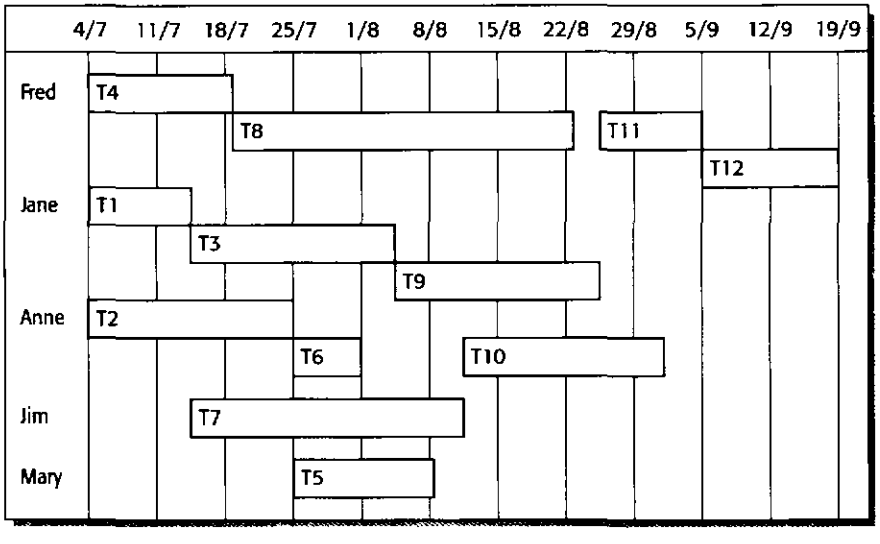


Figura 5.8 Gráfico de asignación de personal/tiempo.

evitar esos riesgos. Los resultados de este análisis de riesgos se deben documentar a lo largo del plan del proyecto junto con el análisis de consecuencias cuando el riesgo ocurra. Identificar éstos y crear planes para minimizar sus efectos en el proyecto se llama gestión de riesgos (Hall, 1998; Ould, 1999).

De forma simple, se puede concebir un riesgo como una probabilidad de que una circunstancia adversa ocurra. Los riesgos son una amenaza para el proyecto, para el software que se está desarrollando y para la organización. Estas categorías de riesgos se definen como se muestra a continuación:

1. *Riesgos del proyecto.* Éstos afectan la calendarización o los recursos del proyecto. Un ejemplo podría ser la pérdida de un diseñador experimentado.
2. *Riesgos del producto.* Éstos afectan a la calidad o al rendimiento del software que se está desarrollando. Un ejemplo podría ser que el rendimiento en un componente que hemos comprado sea menor que el esperado.
3. *Riesgos del negocio.* Éstos afectan a la organización que desarrolla o suministra el software. Por ejemplo, que un competidor introduzca un nuevo producto es un riesgo de negocio.

Por supuesto, estos tipos no son exclusivos entre sí. Si un programador experto abandona el proyecto, esto es un riesgo para el proyecto (debido a que la entrega del sistema se puede retrasar), para el producto (debido a que un sustituto puede no ser tan experto y cometer muchos errores) y para el negocio (debido a que esa experiencia puede no contribuir a negocios futuros).

Los riesgos que pueden afectar a un proyecto dependen del propio proyecto y del entorno organizacional donde se desarrolla. Sin embargo, muchos riesgos son universales. La Figura 5.9 muestra algunos de estos riesgos.

La gestión de riesgos es importante particularmente para los proyectos de software debido a las incertidumbres inherentes con las que se enfrentan muchos proyectos. Estas incerti-

Rotación de personal	**Proyecto**	**Personal con experiencia abandona el proyecto antes de que finalice.**
Cambio de gestión	**Proyecto**	**Habrá un cambio de gestión organizacional con diferentes prioridades.**
No disponibilidad del hardware	**Proyecto**	**El hardware esencial para el proyecto no será entregado a tiempo.**
Cambio de requerimientos	**Proyecto y producto**	**Habrá más cambios en los requerimientos de lo esperado.**
Retrasos en la especificación	**Proyecto y producto**	**Las especificaciones de las interfaces esenciales no estarán a tiempo.**
Subestimación del tamaño	**Proyecto y producto**	**El tamaño del sistema se ha subestimado.**
Bajo rendimiento de la herramienta CASE	**Producto**	**Las herramientas CASE que ayudan al proyecto no tienen el rendimiento esperado.**
Cambio de tecnología	**Negocio**	**Un producto competitivo se pone en venta antes de que el sistema se complete.**
Competencia del producto	**Negocio**	**La tecnología fundamental sobre la que se construirá el sistema se sustituye por nueva tecnología.**

Figura 5.9 Posibles riesgos del software.

dumbres son el resultado de los requerimientos ambiguamente definidos, las dificultades en la estimación de tiempos y los recursos para el desarrollo del software, la dependencia en las habilidades individuales, y los cambios en los requerimientos debidos a los cambios en las necesidades del cliente. Es preciso anticiparse a los riesgos: comprender el impacto de éstos en el proyecto, en el producto y en el negocio, y considerar los pasos para evitarlos. En el caso de que ocurran, se deben crear planes de contingencia para que sea posible aplicar acciones de recuperación.

La Figura 5.10 muestra el proceso de gestión de riesgos. Éste comprende varias etapas:

1. *Identificación de riesgos.* Identificar los posibles riesgos para el proyecto, el producto y los negocios.
2. *Análisis de riesgos.* Valorar las probabilidades y consecuencias de estos riesgos.
3. *Planificación de riesgos.* Crear planes para abordar los riesgos, ya sea para evitarlos o minimizar sus efectos en el proyecto.
4. *Supervisión de riesgos.* Valorar los riesgos de forma constante y revisar los planes para la mitigación de riesgos tan pronto como la información de los riesgos esté disponible.

El proceso de gestión de riesgos, como otros de planificación de proyectos, es un proceso iterativo que se aplica a lo largo de todo el proyecto. Una vez que se genera un conjunto de planes iniciales, se supervisa la situación. En cuanto surja más información acerca de los riesgos, éstos deben analizarse nuevamente y se deben establecer nuevas prioridades. La prevención de riesgos y los planes de contingencia se deben modificar tan pronto como surja nueva información de los riesgos.

Los resultados del proceso de gestión de riesgos se deben documentar en un plan de gestión de riesgos. Éste debe incluir un estudio de los riesgos a los que se enfrenta el proyecto, un análisis de éstos y los planes requeridos para su gestión. Si es necesario, puede incluir algunos resultados de la gestión de riesgos; por ejemplo, planes específicos de contingencia que se activan si aparecen dichos riesgos.

5.4.1 Identificación de riesgos

Ésta es la primera etapa de la gestión de riesgos. Comprende el descubrimiento de los posibles riesgos del proyecto. En principio, no hay que valorarlos o darles prioridad en esta etapa aunque, en la práctica, por lo general no se consideran los riesgos con consecuencias menores o con baja probabilidad.

Esta identificación se puede llevar a cabo a través de un proceso de grupo utilizando un enfoque de tormenta de ideas o simplemente puede basarse en la experiencia. Para ayudar al pro-

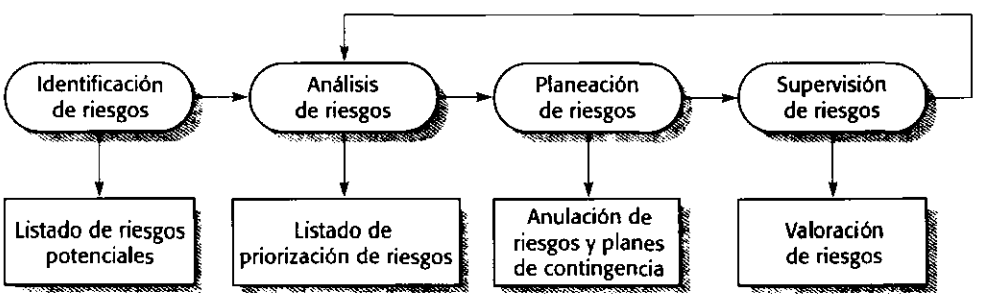


Figura 5.10 El proceso de gestión de riesgos.

ceso, se utiliza una lista de posibles tipos de riesgos. Hay al menos seis tipos de riesgos que pueden aparecer:

1. *Riesgos de tecnología.* Se derivan de las tecnologías de software o de hardware utilizadas en el sistema que se está desarrollando.
2. *Riesgos de personal.* Riesgos asociados con las personas del equipo de desarrollo.
3. *Riesgos organizacionales.* Se derivan del entorno organizacional donde el software se está desarrollando.
4. *Riesgos de herramientas.* Se derivan de herramientas CASE y de otro software de apoyo utilizado para desarrollar el sistema.
5. *Riesgos de requerimientos.* Se derivan de los cambios de los requerimientos del cliente y el proceso de gestionar dicho cambio.
6. *Riesgos de estimación.* Se derivan de los estimados administrativos de las características del sistema y los recursos requeridos para construir dicho sistema.

La Figura 5.11 proporciona algunos ejemplos de riesgos posibles en cada una de estas categorías. El resultado de este proceso debe ser una larga lista de riesgos que podrían presentarse y afectar al producto, al proceso o al negocio.

5.4.2 Análisis de riesgos

Durante este proceso, se considera por separado cada riesgo identificado y se decide acerca de la probabilidad y la seriedad del mismo. No existe una forma fácil de hacer esto —recae en la opinión y experiencia del gestor del proyecto—. No se hace una valoración con números precisos sino en intervalos:

- La probabilidad del riesgo se puede valorar como muy bajo (<10%), bajo (10-25%), moderado (25-50%), alto (50-75%) o muy alto (>75%).
- Los efectos del riesgo pueden ser valorados como catastrófico, serio, tolerable o insignificante.

Tecnología	**La base de datos que se utiliza en el sistema no puede procesar muchas transacciones por segundo como se esperaba.** **Los componentes de software que deben reutilizarse contienen defectos que limitan su funcionalidad.**
Personal	**Es imposible reclutar personal con las habilidades requeridas para el proyecto.** **El personal clave está enfermo y no disponible en momentos críticos.** **La capacitación solicitada para el personal no está disponible.**
Organizacional	**La organización se reestructura de tal forma que una gestión diferente se responsabiliza del proyecto.** **Los problemas financieros de la organización fuerzan a reducciones en el presupuesto del proyecto.**
Herramientas	**Es ineficiente el código generado por las herramientas CASE.** **Las herramientas CASE no se pueden integrar.**
Requerimientos	**Se proponen cambios en los requerimientos que requieren rehacer el diseño.** **Los clientes no comprenden el impacto de los cambios en los requerimientos.**
Estimación	**El tiempo requerido para desarrollar el software está subestimado.** **La tasa de reparación de defectos está subestimada.** **El tamaño del software está subestimado.**

Figura 5.11 Riesgos y tipos de riesgos.

El resultado de este proceso de análisis se debe colocar en una tabla, la cual debe estar ordenada según la seriedad del riesgo. La Figura 5.12 ilustra esto para los riesgos identificados en la Figura 5.11. Obviamente, aquí es arbitraria la valoración de la probabilidad y seriedad. En la práctica, para hacer esta valoración se necesita información detallada del proyecto, el proceso, el equipo de desarrollo y la organización.

Por supuesto, tanto la probabilidad como la valoración de los efectos de un riesgo cambian conforme se disponga de mayor información acerca del riesgo y los planes de gestión del mismo se implementen. Por lo tanto, esta tabla se debe actualizar durante cada iteración del proceso de riesgos.

Una vez que los riesgos se hayan analizado y clasificado, se debe discernir cuáles son los más importantes que se deben considerar durante el proyecto. Este discernimiento debe depender de una combinación de la probabilidad de aparición del riesgo y de los efectos del mismo. En general, siempre se deben tener en cuenta todos los riesgos catastróficos, así como todos los riesgos serios que tienen más que una moderada probabilidad de ocurrir.

Boehm (Boehm, 1988) recomienda identificar y supervisar los «10 riesgos más altos», pero este número parece demasiado arbitrario. El número exacto de riesgos a supervisar debe depender del proyecto. Pueden ser cinco o 15. No obstante, el número apropiado debe ser manejable. Un número muy grande de riesgos requiere obtener mucha información. De los riesgos identificados en la Figura 5.12, conviene considerar los ocho que tienen consecuencias serias o catastróficas.

Los problemas financieros de la organización fuerzan a reducir el presupuesto del proyecto.	Baja	Catastrófico
Es imposible reclutar personal con las habilidades requeridas para el proyecto.	Alta	Catastrófico
El personal clave está enfermo y no disponible en momentos críticos.	Moderada	Serio
Los componentes de software que deben reutilizarse contienen defectos que limitan su funcionalidad.	Moderada	Serio
Se proponen cambios en los requerimientos que requieren rehacer el diseño.	Moderada	Serio
La organización se reestructura de tal forma que cambia el grupo de gestión.	Alta	Serio
La base de datos que se utiliza en el sistema no puede procesar muchas transacciones por segundo como se esperaba.	Moderada	Serio
El tiempo requerido para desarrollar el software está subestimado.	Alta	Serio
Las herramientas CASE no se pueden integrar.	Alta	Tolerable
Los clientes no comprenden el impacto de los cambios en los requerimientos.	Moderada	Tolerable
La capacitación solicitada para el personal no está disponible.	Moderada	Tolerable
La tasa de reparación de defectos está subestimada.	Moderada	Tolerable
El tamaño del software está subestimado.	Alta	Tolerable
Es ineficiente el código generado por las herramientas CASE.	Moderada	insignificante

Figura 5.12 Análisis de riesgos.

5.4.3 Planificación de riesgos

El proceso de planificación de riesgos considera cada uno de los riesgos clave que han sido identificados, así como las estrategias para gestionarlos. Otra vez, no existe un proceso sencillo que nos permita establecer los planes de gestión de riesgos. Depende del juicio y de la experiencia del gestor del proyecto. La Figura 5.13 muestra posibles estrategias para los riesgos que han sido identificados en la Figura 5.12. Estas estrategias seguidas pueden dividirse en tres categorías.

1. *Estrategias de prevención.* Siguiendo estas estrategias, la probabilidad de que el riesgo aparezca se reduce. Un ejemplo de este tipo de estrategias es la estrategia de evitación de defectos en componentes de la Figura 5.13.
2. *Estrategias de minimización.* Siguiendo estas estrategias se reducirá el impacto del riesgo. Un ejemplo de esto es la estrategia frente a enfermedad del personal de la Figura 5.13.
3. *Planes de contingencia.* Seguir estas estrategias es estar preparado para lo peor y tener una estrategia para cada caso. Un ejemplo de este tipo de estrategia es el mostrado en la Figura 5.13 con la estrategia para problemas financieros.

Puede verse aquí una analogía con las estrategias utilizadas en sistemas críticos para asegurar fiabilidad, protección y seguridad. Básicamente, es mejor usar una estrategia para evitar el riesgo. Si esto no es posible, utilizar una para reducir los efectos serios de los riesgos. Finalmente, tener estrategias para reducir el impacto del riesgo en el proyecto y en el producto.

5.4.4 Supervisión de riesgos

La supervisión de riesgos normalmente valora cada uno de los riesgos identificados para decidir si éste es más o menos probable y si han cambiado sus efectos. Por supuesto, esto no se puede observar de forma directa, por lo que se tienen que buscar otros factores para dar indi-

Problemas financieros de la organización	**Preparar un documento breve para el gestor principal que muestre que el proyecto hace contribuciones muy importantes a las metas del negocio.**
Problemas de reclutamiento	**Alertar al cliente de las dificultades potenciales y los posibles retrasos, investigar la compra de componentes.**
Enfermedad de personal	**Reorganizar el equipo de tal forma que haya solapamiento en el trabajo y las personas comprendan el de los demás.**
Componentes defectuosos	**Reemplazar los componentes defectuosos con los comprados de fiabilidad conocida.**
Cambios de los requerimientos	**Rastrear la información para valorar el impacto de los requerimientos, maximizar la información oculta en ellos.**
Reestructuración organizacional	**Preparar un documento breve para el gestor principal que muestre que el proyecto hace contribuciones muy importantes a las metas del negocio.**
Rendimiento de la base de datos	**Investigar la posibilidad de comprar una base de datos de alto rendimiento.**
Tiempo de desarrollo subestimado	**Investigar los componentes comprados y la utilización de un generador de programas.**

Figura 5.13 Estrategia de gestión de riesgos.

cios de la probabilidad del riesgo y sus efectos. Obviamente, estos factores dependen de los tipos de riesgo. La Figura 5.14 da algunos ejemplos de los factores que ayudan en la valoración de estos tipos de riesgos.

La supervisión de riesgos debe ser un proceso continuo y, en cada revisión del progreso de gestión, cada uno de los riesgos clave debe ser considerado y analizado por separado.

Tecnología	Entrega retrasada del hardware o de la ayuda al software, muchos problemas tecnológicos reportados.
Personal	Baja moral del personal, malas relaciones entre los miembros del equipo, disponibilidad de empleo.
Organizacional	Chismorreo organizacional, falta de acciones por el gestor principal.
Herramientas	Rechazo de los miembros del equipo para utilizar herramientas, quejas acerca de las herramientas CASE, peticiones de estaciones de trabajo más potentes.
Requerimientos	Peticiones de muchos cambios en los requerimientos, quejas del cliente.
Estimación	Fracaso en el cumplimiento de los tiempos acordados, y en la eliminación de defectos reportados.

Figura 5.14 Factores de riesgo.

PUNTOS CLAVE

- Es esencial una buena gestión de proyectos de software para que los proyectos de ingeniería de software se desarrollen a tiempo y según presupuesto.
- La gestión de proyectos de software es diferente a la gestión de otro tipo de ingenierías. El software es intangible. Los proyectos pueden ser nuevos o innovadores, por lo que no existe un conjunto de experiencias para guiar su gestión. El proceso del software no se comprende del todo.
- Los gestores de software tienen diversos papeles. Sus actividades más significativas son la planificación, estimación y calendarización de los proyectos. La planificación y la estimación son procesos iterativos. Tienen continuidad a lo largo del proyecto. En cuanto se tenga más información, se deben revisar los planes y calendarios.
- Un hito de un proyecto es el resultado predecible de una actividad en el que se debe presentar un informe del progreso a la gestión. Los hitos ocurren de forma frecuente en un proyecto de software. Una entrega es un hito que se entrega al cliente del proyecto.
- La calendarización de proyectos implica la creación de varias representaciones gráficas de partes del plan del proyecto. Éstas incluyen redes de actividades que muestran las interrelaciones de las actividades del proyecto y gráficos de barras que muestran la duración de dichas actividades.
- Se deben identificar y valorar los riesgos mayores del proyecto para establecer su probabilidad y consecuencias para éste. En cuanto a los riesgos más probables y potencialmente serios, se deben hacer planes para anularlos, gestionarlos o tratarlos. Estos riesgos se deben analizar de manera explícita en cada reunión del progreso del proyecto.

LECTURAS COMPLEMETARIAS

Waltzing with Bears: Managing Risk on Software Projects. Una introducción muy práctica y fácil de leer sobre riesgos y su gestión. (T. DeMarco y T. Lister, 2003, Dorset House.)

Managing Software Quality and Business Risk. El Capítulo 3 de este libro es sencillamente el mejor estudio de riesgos que se haya visto. El libro se orienta a éstos y es probablemente el mejor libro de este tema disponible en el momento de esta redacción. (M. Ould, 1999, John Wiley and Sons.)

The Mythical Man Month. Los problemas de la gestión de software no han cambiado desde los años 60 y éste es uno de los mejores libros sobre el tema. Una recopilación interesante y legible de la gestión de uno de los primeros grandes proyectos de software, el sistema operativo IBM OS/360. La segunda edición incluye otros artículos clásicos de Brooks. (F. P. Brooks, 1975, Addison-Wesley.)

Software Project Survival Guide. Una explicación muy pragmática sobre gestión de software, pero que contiene unos buenos consejos. Es fácil de leer y entender. (S. McConnell, 1998, Microsoft Press.)

Véase la Parte 6 para otras lecturas sobre gestión.

EJERCICIOS

5.1 Explique por qué la intangibilidad de los sistemas de software plantea problemas para la gestión de proyectos de software.

5.2 Explique por qué los mejores programadores no siempre son los mejores gestores de software. La respuesta puede tener como base la lista de actividades de gestión dadas en la Sección 5.1.

5.3 Explique por qué el proceso de planificación de proyectos es iterativo y por qué un plan se debe revisar continuamente durante el proyecto de software.

5.4 Explique brevemente el propósito de cada una de las secciones en un plan de proyecto de software.

5.5 ¿Cuál es la diferencia fundamental entre un hito y una entrega?

5.6 La Figura 5.15 muestra un conjunto de actividades, duraciones y dependencias. Diseñe una red de actividades y un gráfico de barras que muestren la programación del proyecto.

5.7 La Figura 5.5 señala la duración de las tareas para las actividades del proyecto de software. Suponga que hay un serio retraso no anticipado y que en lugar de requerir 10 días, la tarea T5 requiere 40 días. Revise la red de actividades resultante, resaltando el nuevo camino crítico. Diseñe un nuevo gráfico de barras que muestre cómo se podría reorganizar el proyecto.

5.8 Utilizando las instancias referidas en la literatura para los problemas en los proyectos, haga una lista de las dificultades de gestión en esos proyectos de calendarización fallidos. (Comience con el libro de Brooks, sugerido en la sección de lecturas complementarias.)

5.9 Además de los riesgos que se muestran en la Figura 5.11, identifique otros seis posibles riesgos en los proyectos de software.

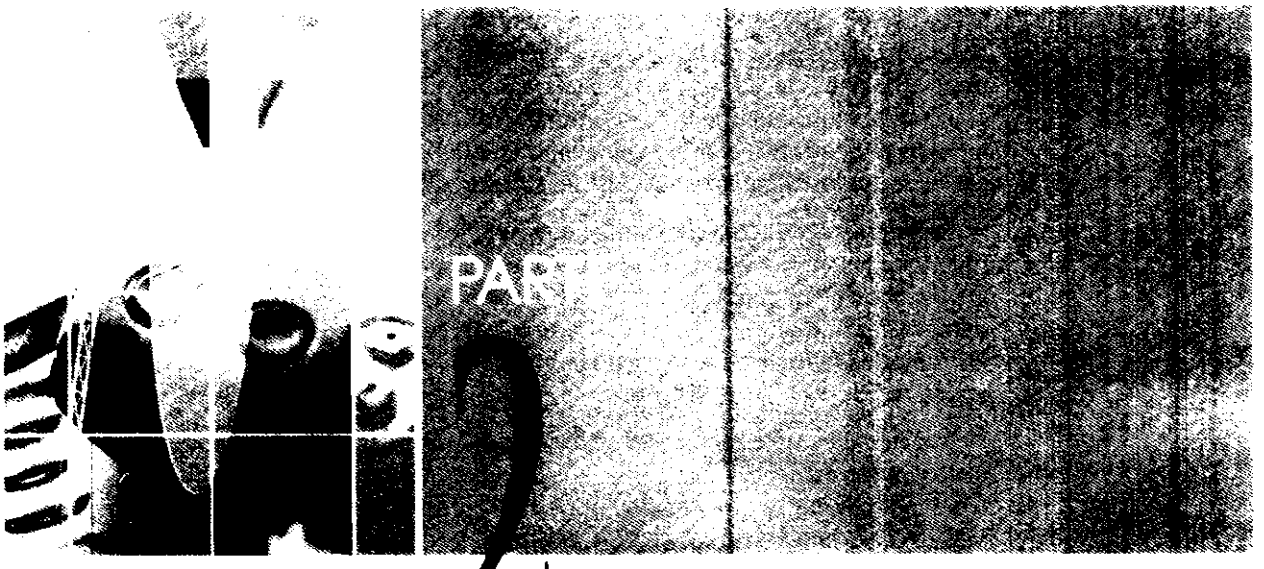
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

Figura 5.15 Duración y dependencias de las tareas.

5.10 Los contratos de precio prefijado, donde el contratista ofrece un precio fijo para completar el sistema, pueden ser utilizados para traspasar los riesgos del proyecto del cliente al contratista. Si algo va mal, el contratista asumirá la diferencia. Indique de qué modo el uso de contratos puede incrementar la probabilidad de la aparición de riesgos.

5.11 Su jefe le ha solicitado que entregue un software en un tiempo que sólo puede ser posible cumplir preguntando al equipo del proyecto si desea trabajar horas extras sin pago alguno. Todos los miembros del equipo tienen hijos pequeños. Comente si debería aceptar esta petición de su jefe o si debería persuadir al equipo para dar su tiempo a la organización más que a sus familias. ¿Qué factores podrían ser significativos en la decisión?

5.12 Como programador, se le ofrece un ascenso como gestor de proyecto, pero su sensación es que puede tener una contribución más efectiva en un papel técnico que en uno administrativo. Comente cuándo debería aceptar ese ascenso.



PARTE 2

REQUERIMIENTOS

Tal vez el principal problema al que nos enfrentamos en el desarrollo de sistemas grandes y complejos es el de la ingeniería de requerimientos. Ésta trata de establecer lo que el sistema debe hacer, sus propiedades emergentes deseadas y esenciales, y las restricciones en el funcionamiento del sistema y los procesos de desarrollo del software. Por lo tanto puede considerar la ingeniería de requerimientos como el proceso de comunicación entre los clientes y usuarios del software y los desarrolladores del mismo.

La ingeniería de requerimientos no es simplemente un proceso técnico. Los requerimientos del sistema están influenciados por las preferencias, aversiones y prejuicios de los usuarios y por cuestiones políticas y organizacionales. Éstas con características fundamentales humanas, y las nuevas tecnologías, como los casos de uso, los escenarios y los métodos formales, no nos ayudan mucho en la resolución de estos problemas espinosos.

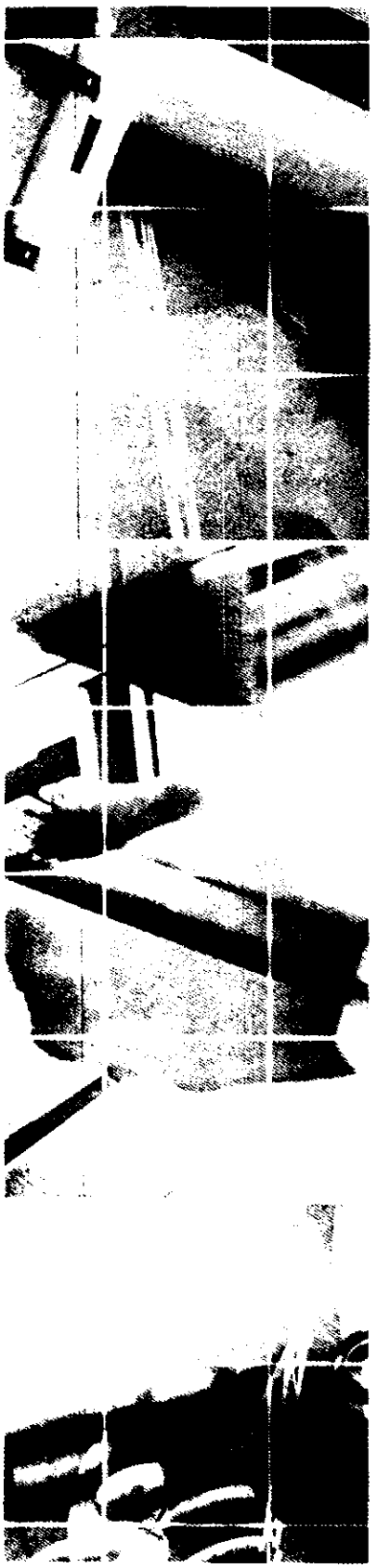
Los capítulos de esta sección se dividen en dos clases —en los Capítulos 6 y 7 introduzco los fundamentos de la ingeniería de requerimientos, y en los Capítulos 8 a 10 describo los modelos y técnicas utilizados en el proceso de ingeniería de requerimientos. Más específicamente:

1. El Capítulo 6 trata sobre los requerimientos del software y los documentos de requerimientos. Se considera qué se entiende por requerimiento, los diferentes tipos de requerimientos y cómo estos requerimientos se organizan en un documento de especificación de requerimientos. En este tema se introduce el segundo caso de estudio, un sistema de biblioteca.

2. El Capítulo 7 se centra en las actividades en el proceso de ingeniería de requerimientos; cómo los estudios de viabilidad siempre deben ser parte de la ingeniería de requerimientos, de las técnicas para la obtención y análisis de requerimientos, y de la validación de requerimientos. Debido a que los requerimientos inevitablemente cambian, también se aborda el importante tema de la gestión de requerimientos.

3. El Capítulo 8 describe los tipos de modelos de sistemas que se pueden desarrollar en el proceso de ingeniería de requerimientos. Éstos proporcionan una descripción más detallada a los desarrolladores del sistema. Aquí el énfasis está en el modelado orientado a objetos pero también incluyo una descripción de los diagramas de flujo de datos. Se considera que éstos son intuitivos y útiles, especialmente para darle una imagen de cómo la información es procesada por un sistema desde el principio hasta el final.

4. El énfasis en los Capítulos 9 y 10 está en la especificación de los sistemas críticos. El Capítulo 9 aborda la especificación de las propiedades de confiabilidad emergentes. Describe los enfoques dirigidos por riesgos y asuntos de la especificación de la seguridad, la fiabilidad y la protección. En el Capítulo 10, se introducen las técnicas de especificación formal. Los métodos formales han tenido menos impacto del que se predijo, pero cada vez se utilizan más en la especificación de la seguridad y de los sistemas de misión críticos. Aborda tanto los enfoques algebraicos como los basados en modelos en este capítulo.



6 Requerimientos del software

Objetivos

Los objetivos de este capítulo son presentar los requerimientos de sistemas software y explicar las diferentes formas de expresar los requerimientos del software. Cuando haya leído este capítulo:

- entenderá los conceptos de requerimientos del usuario y del sistema y por qué estos requerimientos se deben escribir de diferentes formas;
- entenderá las diferencias entre los requerimientos del software funcionales y no funcionales;
- entenderá cómo los requerimientos se pueden organizar en un documento de requerimientos del software.

Contenidos

Los requerimientos para un sistema son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas. Estos requerimientos reflejan las necesidades de los clientes de un sistema que ayude a resolver algún problema como el control de un dispositivo, hacer un pedido o encontrar información. El proceso de descubrir, analizar, documentar y verificar estos servicios y restricciones se denomina *ingeniería de requerimientos* (RE). Este capítulo se centra en dichos requerimientos y cómo describirlos. En el Capítulo 4 se dio una introducción al proceso de ingeniería de requerimientos y en el Capítulo 7 se analizará con más detalle.

El término *requerimiento* no se utiliza de una forma constante en la industria de software. En algunos casos, un requerimiento es simplemente una declaración abstracta de alto nivel de un servicio que debe proporcionar el sistema o una restricción de éste. En el otro extremo, es una definición detallada y formal de una función del sistema. Davis (Davis, 1993) explica por qué existen estas diferencias:

> Si una compañía desea establecer un contrato para un proyecto de desarrollo de software grande, debe definir sus necesidades de una forma suficientemente abstracta para establecer a partir de ella una solución. Los requerimientos deben redactarse de tal forma que varios contratistas pueden licitar el contrato, ofreciendo, quizás, formas diferentes de cumplir las necesidades de los clientes en la organización. Una vez que el contrato se asigna, el contratista debe redactar una definición del sistema para el cliente más detalladamente de forma que éste comprenda y pueda validar lo que hará el software. Ambos documentos se pueden denominar documento de requerimientos para el sistema.

Algunos de los problemas que surgen durante el proceso de ingeniería de requerimientos son resultado de no hacer una clara separación entre estos diferentes niveles de descripción. Aquí se distinguen utilizando la denominación *requerimientos del usuario* para designar los requerimientos abstractos de alto nivel, y *requerimientos del sistema* para designar la descripción detallada de lo que el sistema debe hacer. Los requerimientos del usuario y del sistema se pueden definir como se muestra a continuación:

1. *Los requerimientos del usuario* son declaraciones, en lenguaje natural y en diagramas, de los servicios que se espera que el sistema proporcione y de las restricciones bajo las cuales debe funcionar.
2. *Los requerimientos del sistema* establecen con detalle las funciones, servicios y restricciones operativas del sistema. El documento de requerimientos del sistema (algunas veces denominado especificación funcional) debe ser preciso. Debe definir exactamente qué es lo que se va a implementar. Puede ser parte del contrato entre el comprador del sistema y los desarrolladores del software.

Diferentes niveles de especificación del sistema son de utilidad debido a que comunican la información del sistema a diferentes tipos de lectores. La Figura 6.1 ilustra la distinción entre los requerimientos del usuario y del sistema. Este ejemplo de un sistema de biblioteca muestra la manera en que un requerimiento del usuario se expande en varios requerimientos del sistema. Puede verse en la Figura 6.1 que el requerimiento del usuario es más abstracto, y que los requerimientos del sistema añaden detalle, explicando los servicios y funciones que deben ser proporcionados por el sistema a desarrollar.

Es necesario redactar los requerimientos en diversos niveles de detalle debido a que diferentes tipos de lectores los utilizan de distinta manera. La Figura 6.2 muestra los tipos de lectores de los requerimientos de usuario y del sistema. Los lectores de los requerimientos de usuario normalmente no tratan de cómo se implementará el sistema y pueden ser administradores que no están interesados en los recursos detallados del sistema. Los lectores de los re-

Definición del requerimiento del usuario

1. LIBSYS controlará todos los datos requeridos por las agencias que licencian los derechos de autor en el Reino Unido y en otra parte.

Especificación de los requerimientos del sistema

1.1 Al hacer una petición de un documento del LIBSYS, el solicitante se presentará con un formulario que registre los detalles del usuario y de la petición hecha.
1.2 El formulario de petición del LIBSYS será almacenado en el sistema durante cinco años desde la fecha de la petición.
1.3 Todos los formularios de peticiones del LIBSYS se deben indexar por usuario, por el nombre del material solicitado y por el proveedor de la petición.
1.4 El LIBSYS mantendrá un fichero en el que se registrarán todas las peticiones que se han hecho al sistema.
1.5 Para el material donde se aplican los derechos de préstamo del autor, los detalles del préstamo serán enviados mensualmente a las agencias que licencian los derechos de autor que se han registrado con LIBYSS.

Figura 6.1 Requerimientos del usuario y del sistema.

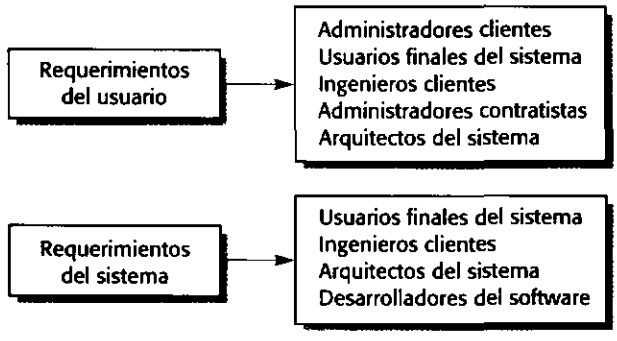


Figura 6.2 Lectores de los diferentes tipos de especificaciones.

querimientos del sistema necesitan saber con más precisión qué hará el sistema debido a que están interesados en cómo ayudará esto a los procesos de negocio o debido a que están implicados en la implementación del sistema.

6.1 Requerimientos funcionales y no funcionales

A menudo, los requerimientos de sistemas software se clasifican en funcionales y no funcionales, o como requerimientos del dominio:

1. *Requerimientos funcionales*. Son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares. En algunos casos, los requerimientos funcionales de los sistemas también pueden declarar explícitamente lo que el sistema no debe hacer.
2. *Requerimientos no funcionales*. Son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo y

estándares. Los requerimientos no funcionales a menudo se aplican al sistema en su totalidad. Normalmente apenas se aplican a características o servicios individuales del sistema.

3. *Requerimientos del dominio*. Son requerimientos que provienen del dominio de aplicación del sistema y que reflejan las características y restricciones de ese dominio. Pueden ser funcionales o no funcionales.

En realidad, la distinción entre diferentes tipos de requerimientos no es tan clara como sugieren estas definiciones. Por ejemplo, un requerimiento del usuario sobre seguridad podría parecer un requerimiento no funcional. Sin embargo, cuando se desarrolla en detalle, puede generar otros requerimientos que son claramente funcionales, como la necesidad de incluir en el sistema recursos para la autentificación del usuario.

6.1.1 Requerimientos funcionales

Los requerimientos funcionales de un sistema describen lo que el sistema debe hacer. Estos requerimientos dependen del tipo de software que se desarrolle, de los posibles usuarios del software y del enfoque general tomado por la organización al redactar requerimientos. Cuando se expresan como requerimientos del usuario, habitualmente se describen de una forma bastante abstracta. Sin embargo, los requerimientos funcionales del sistema describen con detalle la función de éste, sus entradas y salidas, excepciones, etcétera.

Los requerimientos funcionales para un sistema software se pueden expresar de diferentes formas. A continuación se presentan algunos ejemplos de estos requerimientos funcionales para un sistema de biblioteca universitario, denominado LIBSYS, utilizado por estudiantes y personal docente que solicitan libros y documentos de otras bibliotecas.

1. El usuario deberá tener la posibilidad de buscar en el conjunto inicial de la base de datos o seleccionar un subconjunto de ella.
2. El sistema deberá proporcionar visores adecuados para que el usuario lea documentos en el almacén de documentos.
3. A cada pedido se le deberá asignar un identificador único (ID_PEDIDO), que el usuario podrá copiar al área de almacenamiento permanente de la cuenta.

Estos requerimientos funcionales del usuario definen los recursos específicos que el sistema debe proporcionar. Dichos requerimientos se toman del documento de requerimientos del usuario, e ilustran los diferentes niveles de detalle en que se pueden redactar los requerimientos funcionales (contraste los requerimientos 1 y 3).

El sistema LIBSYS es una interfaz única para diferentes bases de datos de artículos. Esto permite a los usuarios descargar copias de artículos publicados en revistas, periódicos y publicaciones científicas. Una descripción más detallada de los requerimientos para el sistema en el cual se basa LIBSYS se puede ver en mi libro con Gerald Kotonya sobre ingeniería de requerimientos (Kontonya y Sommerville, 1998).

La impresión en la especificación de requerimientos es la causa de muchos de los problemas de la ingeniería del software. Para un desarrollador de sistemas es natural dar interpretaciones de un requerimiento ambiguo con el fin de simplificar su implementación. Sin embargo, a menudo no es lo que el cliente desea. Se deben establecer nuevos requerimientos y hacer cambios en el sistema. Por supuesto, esto retrasa la entrega de éste e incrementa los costes.

Considere el segundo ejemplo de los requerimientos para el sistema de biblioteca que se refiere a los «visores adecuados» que debe proporcionar el sistema. El sistema de biblioteca

puede mostrar documentos en diferentes formatos; la intención de este requerimiento es que los visores para todos estos formatos estén disponibles. Sin embargo, el requerimiento está ambiguamente redactado; no clarifica que se deben proporcionar los visores de cada formato. Un desarrollador bajo la presión del tiempo sencillamente podría proporcionar un visor de texto y afirmar que se ha cumplido el requerimiento.

En principio, la especificación de requerimientos funcionales de un sistema debe estar completa y ser consistente. La completitud significa que todos los servicios solicitados por el usuario deben estar definidos. La consistencia significa que los requerimientos no deben tener definiciones contradictorias. En la práctica, para sistemas grandes y complejos, es prácticamente imposible alcanzar los requerimientos de consistencia y completitud.

Una razón de esto es que es fácil cometer errores y omisiones cuando se redactan especificaciones para sistemas grandes y complejos. Otra razón es que los *stakeholders* del sistema (véase el Capítulo 7) tienen necesidades diferentes, y a menudo contradictorias. Estas contradicciones pueden no ser obvias cuando los requerimientos se especifican por primera vez, por lo que se incluyen requerimientos contradictorios en la especificación. Es posible que los problemas surjan solamente después de un análisis más profundo o, a veces, después de que se termine el desarrollo y el sistema se entregue al cliente.

6.1.2 Requerimientos no funcionales

Los requerimientos no funcionales, como su nombre sugiere, son aquellos requerimientos que no se refieren directamente a las funciones específicas que proporciona el sistema, sino a las propiedades emergentes de éste como la fiabilidad, el tiempo de respuesta y la capacidad de almacenamiento. De forma alternativa, definen las restricciones del sistema como la capacidad de los dispositivos de entrada/salida y las representaciones de datos que se utilizan en las interfaces del sistema.

Los requerimientos no funcionales rara vez se asocian con características particulares del sistema. Más bien, estos requerimientos especifican o restringen las propiedades emergentes del sistema, como se explicó en el Capítulo 2. Por lo tanto, pueden especificar el rendimiento del sistema, la protección, la disponibilidad, y otras propiedades emergentes. Esto significa que a menudo son más críticos que los requerimientos funcionales particulares. Los usuarios del sistema normalmente pueden encontrar formas de trabajar alrededor de una función del sistema que realmente no cumple sus necesidades. Sin embargo, el incumplimiento de un requerimiento no funcional puede significar que el sistema entero sea inutilizable. Por ejemplo, si un sistema de vuelo no cumple sus requerimientos de fiabilidad, no se certificará como seguro para el funcionamiento; si un sistema de control de tiempo real no cumple sus requerimientos de rendimiento, las funciones de control no funcionarán correctamente.

Los requerimientos no funcionales no sólo se refieren al sistema software a desarrollar. Algunos de estos requerimientos pueden restringir el proceso que se debe utilizar para desarrollar el sistema. Ejemplos de requerimientos de procesos son la especificación de los estándares de calidad que se deben utilizar en el proceso, una especificación que el diseño debe producir con una herramienta CASE particular y una descripción del proceso a seguir.

Los requerimientos no funcionales surgen de las necesidades del usuario, debido a las restricciones en el presupuesto, a las políticas de la organización, a la necesidad de interoperabilidad con otros sistemas software o hardware, o a factores externos como regulaciones de segu-

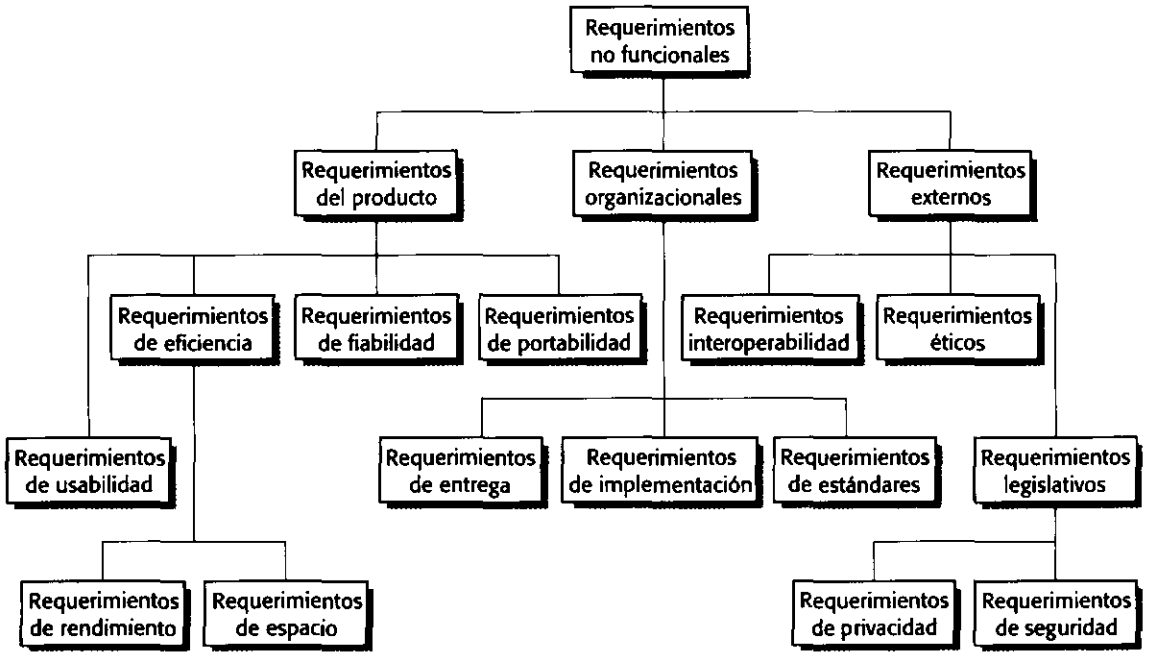


Figura 6.3 Tipos de requerimientos no funcionales.

ridad o legislaciones sobre privacidad. La Figura 6.3 es una clasificación de los requerimientos no funcionales. Puede verse en este diagrama que los requerimientos no funcionales pueden venir de las características requeridas del software (requerimientos del producto), de la organización que desarrolla el software (requerimientos organizacionales) o de fuentes externas.

Los tipos de requerimientos no funcionales son:

1. *Requerimientos del producto.* Estos requerimientos especifican el comportamiento del producto. Algunos ejemplos son los requerimientos de rendimiento en la rapidez de ejecución del sistema y cuánta memoria se requiere; los requerimientos de fiabilidad que fijan la tasa de fallos para que el sistema sea aceptable; los requerimientos de portabilidad, y los requerimientos de usabilidad.
2. *Requerimientos organizacionales.* Estos requerimientos se derivan de políticas y procedimientos existentes en la organización del cliente y en la del desarrollador. Algunos ejemplos son los estándares en los procesos que deben utilizarse; los requerimientos de implementación, como los lenguajes de programación o el método de diseño a utilizar, y los requerimientos de entrega que especifican cuándo se entregará el producto y su documentación.
3. *Requerimientos externos.* Este gran apartado incluye todos los requerimientos que se derivan de los factores externos al sistema y de su proceso de desarrollo. Éstos pueden incluir los requerimientos de interoperabilidad que definen la manera en que el sistema interactúa con sistemas de otras organizaciones; los requerimientos legislativos que deben seguirse para asegurar que el sistema funcione dentro de la ley, y los requerimientos éticos. Estos últimos son puestos en un sistema para asegurar que será aceptado por sus usuarios y por el público en general.

La Figura 6.4 muestra ejemplos de requerimientos del producto, organizacionales y externos tomados del sistema de biblioteca LIBSYS cuyos requerimientos del usuario se analiza-

Requerimiento del producto
8.1 La interfaz de usuario del LIBSYS se implementará como HTML simple sin marcos o applets Java.

Requerimiento organizacional
9.3.2 El proceso de desarrollo del sistema y los documentos a entregar deberán ajustarse al proceso y a los productos a entregar definidos en XYZCo-SP-STAN-95.

Requerimiento externo
10.6 El sistema no deberá revelar al personal de la biblioteca que lo utilice ninguna información personal de los usuarios del sistema aparte de su nombre y número de referencia de la biblioteca.

Figura 6.4 Ejemplos requerimientos no funcionales.

ron en la Sección 6.1.1. El requerimiento del producto restringe la libertad de los diseñadores del LIBSYS en la implementación de la interfaz de usuario del sistema. No dice nada acerca de la funcionalidad del LIBSYS e identifica claramente una restricción del sistema más que una función. Se incluye este requerimiento debido a que simplifica el problema de asegurar que el sistema funcione con navegadores diferentes.

El requerimiento organizacional especifica que el sistema se debe desarrollar conforme a un proceso estándar de la compañía definido como XYZCo-SP-STAN-95. El requerimiento externo se deriva de la necesidad del sistema de cumplir la legislación sobre privacidad. Especifica que no se debe permitir al personal de la biblioteca el acceso a datos, como la dirección de los usuarios del sistema, que no necesitan para realizar su trabajo.

Un problema común con los requerimientos no funcionales es que pueden ser difíciles de verificar. Los usuarios o clientes declaran a menudo estos requerimientos como metas generales tales como la facilidad de uso, la capacidad del sistema para recuperarse de los fallos o la respuesta rápida al usuario. Estas metas imprecisas causan problemas a los desarrolladores del sistema puesto que dejan abierta la posibilidad a interpretación, lo que provoca discusiones subsecuentes una vez que el sistema se entrega. Como ilustración de este problema, considere la Figura 6.5. Esta figura muestra una meta del sistema que se refiere a la usabilidad de un sistema de control del tráfico aéreo y es típico de cómo un usuario puede expresar los requerimientos de usabilidad. Se ha reescrito para mostrar la manera en que la meta se puede expresar como un requerimiento no funcional que se pueda probar. Aunque es imposible verificar objetivamente la meta del sistema, se pueden diseñar pruebas del sistema para contar los errores cometidos por los controladores utilizando un simulador del sistema.

Siempre que sea posible, se deben redactar los requerimientos funcionales de manera cuantitativa para que se puedan probar de un modo objetivo. La Figura 6.6 muestra varias métricas posibles que pueden usarse para especificar las propiedades no funcionales del sistema. Se pueden medir estas características cuando se prueba el sistema para comprobar si cumple sus requerimientos no funcionales.

Una meta del sistema
Debe ser fácil para los controladores experimentados utilizar el sistema y se debe organizar de tal modo que se minimicen los errores del usuario.

Un requerimiento no funcional verificable
Después de una formación de dos horas, a los controladores experimentados les deberá ser posible utilizar todas las funciones del sistema. Después de esta formación, la media de errores cometidos por los usuarios experimentados no excederá de dos por día.

Figura 6.5 Metas del sistema y requerimientos verificables.

Rapidez	**Transacciones procesadas por segundo.** **Tiempo de respuesta al usuario y a eventos** **Tiempo de actualización de la pantalla**
Tamaño	**K Bytes** **Número de chips de RAM**
Facilidad de uso	**Tiempo de formación** **Número de cuadros de ayuda**
Fiabilidad	**Tiempo medio entre fallos** **Probabilidad de no disponibilidad** **Tasa de ocurrencia de fallos** **Disponibilidad**
Robustez	**Tiempo de reinicio después de fallos** **Porcentaje de eventos que provocan fallos** **Probabilidad de corrupción de los datos después de fallos**
Portabilidad	**Porcentaje de declaraciones dependientes del objetivo** **Número de sistemas objetivo**

Figura 6.6 Métricas para especificar requerimientos no funcionales.

En la práctica, sin embargo, los clientes de un sistema pueden encontrar prácticamente imposible traducir sus metas en requerimientos cuantitativos. Para algunas metas, como las de mantenibilidad, no existen métricas que se puedan utilizar. En otros casos, aun cuando sea posible la especificación cuantitativa, es posible que los clientes no puedan relacionar sus necesidades con estas especificaciones. No entienden lo que significa un cierto número que define la fiabilidad requerida (por ejemplo) en función de su experiencia diaria con los sistemas informáticos. Además, el coste de verificar de forma objetiva los requerimientos no funcionales cuantitativos puede ser muy alto, y los clientes que pagan el sistema quizás piensen que estos costes no son justificados.

Por lo tanto, los documentos de los requerimientos a menudo incluyen las declaraciones de las metas mezcladas con los requerimientos. Dichas metas pueden ser útiles para los desarrolladores puesto que dan idea de las prioridades del cliente. Sin embargo, siempre se les debe advertir que están abiertas a interpretaciones erróneas y que no se pueden verificar de forma objetiva.

A menudo, los requerimientos no funcionales entran en conflicto e interactúan con otros requerimientos funcionales o no funcionales. Por ejemplo, puede haber un requerimiento que especifique que la capacidad máxima de memoria utilizada por un sistema no debe ser mayor de 4 Mbytes. Las restricciones de memoria son comunes en los sistemas embebidos donde el espacio y el peso están limitados y se debe minimizar el número de chips de ROM que almacenan el sistema software. Otro requerimiento podría ser que el sistema debe codificarse utilizando Ada, un lenguaje de programación para el desarrollo de software crítico de tiempo real. Sin embargo, puede que no sea posible compilar con menos de 4 Mbytes un programa en Ada con la funcionalidad requerida. Por lo tanto, tiene que haber un equilibro entre estos requerimientos: un lenguaje alternativo de desarrollo o un aumento en la memoria del sistema.

Es de utilidad diferenciar los requerimientos funcionales y no funcionales en el documento de requerimientos. En la práctica, esto es difícil. Si los requerimientos no funcionales se declaran de forma separada de los funcionales, algunas veces es difícil ver la relación entre ellos. Si se declaran con los requerimientos funcionales, puede resultar difícil separar las con-

sideraciones funcionales y no funcionales e identificar los requerimientos que se refieren al sistema como un todo. Sin embargo, se deben resaltar explícitamente los requerimientos que claramente se refieran a las propiedades emergentes del sistema, como el rendimiento o la fiabilidad. Se puede hacer colocándolos en una sección aparte o diferenciándolos, de alguna forma, de los otros requerimientos del sistema.

Los requerimientos no funcionales como los requerimientos de seguridad y protección son especialmente importantes en los sistemas críticos. Por lo tanto, se tratan con mayor detalle los requerimientos de confiabilidad en el Capítulo 9, el cual incluye la especificación de los sistemas críticos.

6.1.3 Los requerimientos del dominio

Los requerimientos del dominio se derivan del dominio de aplicación del sistema más que de las necesidades específicas de los usuarios. Normalmente incluyen terminología especializada del dominio o referencias a conceptos del dominio. Pueden ser requerimientos funcionales nuevos, restringir los existentes o establecer cómo se deben ejecutar cálculos particulares. Debido a que éstos se especializan, a los ingenieros de software a menudo les resulta difícil entender cómo se relacionan con los otros requerimientos del sistema.

Los requerimientos del dominio son importantes debido a que a menudo reflejan los fundamentos del dominio de aplicación. Si estos requerimientos no se satisfacen, puede ser imposible hacer que el sistema funcione de forma satisfactoria. El sistema LIBSYS incluye varios requerimientos del dominio:

1. Deberá existir una interfaz de usuario estándar para todas las bases de datos que estará basada en el estándar Z39.50.
2. Debido a las restricciones en los derechos de autor, algunos documentos deberán borrarse inmediatamente después de su llegada. Dependiendo de los requerimientos del usuario, estos documentos se imprimirán de forma local en el servidor del sistema para ser distribuidos de forma manual al usuario o se enviarán a la impresora de la red.

El primer requerimiento es una restricción de diseño. Establece que la interfaz de usuario para la base de datos debe implementarse según un estándar bibliotecario específico. Los desarrolladores, por lo tanto, tienen que informarse sobre el estándar antes de empezar el diseño de la interfaz. El segundo requerimiento se introduce debido a las leyes de derechos de autor que se aplican a los materiales utilizados en las bibliotecas. Establece que el sistema debe incluir un recurso automático para borrar algunas clases de documentos al ser impresos. Esto significa que los usuarios del sistema de biblioteca no pueden tener su propia copia electrónica del documento.

Para ilustrar los requerimientos del dominio que especifican cómo se lleva a cabo un cálculo considere la Figura 6.7, tomada de la especificación de un sistema de protección automatizada de trenes. Este sistema detiene de forma automática un tren si pasa por una señal roja. Este requerimiento establece la manera en que dicho sistema calcula la desacele-

La desaceleración del tren se calculará como:

$$D_{tren} = D_{control} + D_{gradiente}$$

donde $D_{gradiente}$ es 9,81 ms^2 * gradiente compensado/alfa y donde los valores de 9,81 ms^2/alfa se conocen para diferentes tipos de trenes.

Figura 6.7 Un requerimiento del dominio obtenido de un sistema de protección de trenes.

ración del tren. La terminología utilizada es específica del dominio. Para entenderla, se necesita una cierta comprensión del funcionamiento del sistema ferroviario y las características de los trenes.

El requerimiento para el sistema de trenes ilustra el problema principal con los requerimientos del dominio. Éstos se escriben en el lenguaje del dominio de aplicación (ecuaciones matemáticas en este caso), y a menudo es difícil de comprender por los ingenieros de software. Los expertos en el dominio pueden dejar fuera de un requerimiento información sencillamente porque para ellos es obvia. Sin embargo, puede no serlo para los desarrolladores del sistema y, por lo tanto, es posible que implementen el requerimiento de forma equivocada.

6.2 Requerimientos del usuario

Los requerimientos del usuario para un sistema deben describir los requerimientos funcionales y no funcionales de tal forma que sean comprensibles por los usuarios del sistema sin conocimiento técnico detallado. Únicamente deben especificar el comportamiento externo del sistema y deben evitar, tanto como sea posible, las características de diseño del sistema. Por consiguiente, si se están redactando requerimientos del usuario, no se debe utilizar jerga del software, notaciones estructuradas o formales, o describir los requerimientos por la descripción de la implementación del sistema. Deben redactarse en un lenguaje sencillo, con tablas y formularios sencillos y diagramas intuitivos.

Sin embargo, pueden surgir diversos problemas cuando se redactan con frases del lenguaje natural en un documento de texto:

1. *Falta de claridad.* Algunas veces es difícil utilizar el lenguaje de forma precisa y no ambigua sin hacer el documento poco conciso y difícil de leer.
2. *Confusión de requerimientos.* No se distinguen claramente los requerimientos funcionales y no funcionales, las metas del sistema y la información para el diseño.
3. *Conjunción de requerimientos.* Diversos requerimientos diferentes se pueden expresar de forma conjunta como un único requerimiento.

Para ilustrar algunos de estos problemas, considere uno de los requerimientos para la biblioteca que se muestran en la Figura 6.8.

Este requerimiento incluye tanto información conceptual como detallada. Expresa que debe hacer un sistema de contabilidad como una parte inherente del LIBSYS. Sin embargo, también incluye el detalle de que el sistema de contabilidad debe admitir descuentos para los usuarios habituales del LIBSYS. Este detalle podría ubicarse mejor en la especificación de requerimientos del sistema.

Es una buena práctica separar en el documento de requerimientos los requerimientos del usuario de los detallados del sistema. Por otra parte, los lectores no técnicos de los requerimientos del usuario se pueden confundir con detalles que sólo son relevantes a los lectores técnicos. La Figura 6.9 ilustra esta confusión. Este ejemplo se ha tomado de un documento de requerimientos real para una herramienta CASE para editar modelos de diseño de software.

4.5 El LIBSYS proporcionará un sistema de contabilidad financiera que mantendrá registro de todos los pagos hechos por los usuarios del sistema. Los administradores del sistema pueden configurar este sistema de forma que los usuarios habituales puedan recibir un precio rebajado.

Figura 6.8 Un requerimiento de usuario para un sistema de contabilidad en el LIBSYS.

2.6 Recursos de la cuadrícula. Para ayudar a la ubicación de entidades en un diagrama, el usuario puede activar una cuadrícula en centímetros o en pulgadas, mediante una opción en el panel de control. Al principio, la cuadrícula está desactivada. Esta cuadrícula se puede activar o desactivar en cualquier momento durante una sesión de edición y poner en pulgadas y centímetros. La opción de cuadrícula se proporcionará en la vista de reducción de ajuste, pero el número de líneas de la cuadrícula a mostrar se reducirá para evitar saturar el diagrama más pequeño con líneas de cuadrícula.

Figura 6.9 Un requerimiento de usuario para un editor de cuadrícula.

El usuario puede especificar que se debe mostrar una cuadrícula para que las entidades se coloquen de forma precisa en un diagrama.

La primera frase mezcla tres diferentes clases de requerimientos.

1. Un requerimiento funcional conceptual que establece que el sistema de edición debe proporcionar una cuadrícula. Se presenta la justificación de esto.
2. Un requerimiento no funcional que proporciona información detallada de las unidades de la cuadrícula (centímetros o pulgadas).
3. Un requerimiento de interfaz de usuario no funcional que define la manera en que la cuadrícula es activada o desactivada por el usuario.

El requerimiento de la Figura 6.9 también muestra una parte de la información de inicialización. Define que la cuadrícula está inicialmente desactivada. Sin embargo, no define sus unidades cuando se activa. Proporciona alguna información detallada —como la de intercambiar las unidades, pero no el espaciado entre las líneas de la cuadrícula.

Los requerimientos del usuario que incluyen demasiada información restringen la libertad del desarrollador del sistema para proporcionar soluciones innovadoras a los problemas del usuario y son difíciles de comprender. Los requerimientos del usuario deben simplemente enfocarse a los recursos principales que se deben proporcionar. Se ha redactado nuevamente el requerimiento para el editor de la cuadrícula (Figura 6.10) para enfocarlo solamente en las características esenciales del sistema.

Cuando sea posible, se debe intentar asociar un fundamento con cada requerimiento del usuario. El fundamento debe explicar por qué se ha incluido el requerimiento y es particularmente útil cuando cambian éstos. Por ejemplo, el fundamento en la Figura 6.10 reconoce que es de utilidad que una cuadrícula activa sitúe automáticamente objetos en una línea de la cuadrícula. Sin embargo, esto se rechaza de forma deliberada en favor de una ubicación manual. Si en alguna etapa posterior se propone un cambio a esto, entonces estará claro que la utilización de una cuadrícula pasiva fue algo premeditado más que una decisión de implementación.

2.6.1 Recursos de la cuadrícula

El editor proporcionará un recurso para la cuadrícula donde una matriz de líneas horizontales y verticales proporciona un fondo para la ventana del editor. Esta cuadrícula será pasiva, donde la alineación de entidades es responsabilidad del usuario.

> *Fundamento:* Una cuadrícula ayuda al usuario a crear un diagrama ordenado con entidades bien espaciadas. Aunque en una cuadrícula activa pueda ser de utilidad que las entidades se ajusten a las líneas de la cuadrícula, la ubicación es imprecisa. El usuario es la mejor persona para decidir dónde se deberían ubicar las entidades.

Especificación: ECLIPSE/WS/Herramientas/DE/FS Sección 5.6

Fuente: Ray Wilson, Glasgow Office

Figura 6.10 Una definición de un recurso para el editor de cuadrícula.

Para minimizar los malentendidos al redactar los requerimientos del usuario, se recomienda seguir algunas pautas sencillas:

1. Inventar un formato estándar y asegurar que todos los requerimientos se adhieren al formato. Estandarizar el formato hace que las omisiones sean menos probables y los requerimientos más fáciles de verificar. El formato utilizado muestra el requerimiento inicial en negrita, incluyendo una declaración del fundamento con cada requerimiento del usuario y una referencia a la especificación más detallada de los requerimientos del sistema. También se puede incluir información sobre quién propuso el requerimiento (la fuente del requerimiento), de modo que se sepa a quién consultar si se tiene que cambiar el requerimiento.
2. Utilizar el lenguaje de forma consistente. Siempre debe distinguir entre los requerimientos deseables y los obligatorios. Los *requerimientos obligatorios* son los requerimientos a los que el sistema debe dar soporte y normalmente se redactan en futuro simple. Los *requerimientos deseables* no son fundamentales y se redactan en futuro condicional.
3. Resaltar el texto (con negrita, cursiva o color) para distinguir las partes clave del requerimiento.
4. Evitar, hasta donde sea posible, el uso de jerga informática. Sin embargo, inevitablemente se incluirán términos técnicos detallados en los requerimientos del usuario.

Los Robertson (Roberton y Robertson, 1999), en su libro que estudia el método de ingeniería de requerimientos VOLERE, recomiendan que los requerimientos del usuario sean redactados inicialmente en tarjetas, un requerimiento por tarjeta. Sugieren varios campos en cada tarjeta, como el fundamento de los requerimientos, las dependencias con otros requerimientos, la fuente de los requerimientos, el material de apoyo, etcétera. Esto extiende el formato utilizado en la Figura 6.10, y se puede emplear tanto para requerimientos del usuario como del sistema.

6.3 Requerimientos del sistema

Los requerimientos del sistema son versiones extendidas de los requerimientos del usuario que son utilizados por los ingenieros de software como punto de partida para el diseño del sistema. Agregan detalle y explican cómo el sistema debe proporcionar los requerimientos del usuario. Pueden ser utilizados como parte del contrato para la implementación del sistema y, por lo tanto, deben ser una especificación completa y consistente del sistema entero.

En teoría, los requerimientos del sistema simplemente deben describir el comportamiento externo del sistema y sus restricciones operativas. No deben tratar de cómo se debe diseñar o implementar el sistema. Sin embargo, en el nivel de detalle requerido para especificar completamente un sistema software complejo, es imposible, en la práctica, excluir toda la información de diseño. Existen varias razones para esto:

1. Puede tener que diseñar una arquitectura inicial del sistema para ayudar a estructurar la especificación de requerimientos. Los requerimientos del sistema se organizan conforme a los diferentes subsistemas que componen el sistema. Como se expone en los Capítulos 7 y 18, esta definición arquitectónica es esencial si quiere reutilizar componentes software cuando implementa el sistema.
2. En muchos casos, los sistemas deben interoperar con otros ya existentes. Esto restringe el diseño, y estas restricciones imponen requerimientos en el sistema nuevo.

3. Es necesario el uso de una arquitectura específica para satisfacer los requerimientos no funcionales (como la programación por *n* versiones para conseguir fiabilidad, comentada en el Capítulo 20). Un regulador externo que necesita certificar que el sistema es seguro puede especificar que un diseño arquitectónico que ya ha sido certificado sea utilizado.

A menudo se utiliza el lenguaje natural para redactar, además de los requerimientos del usuario, las especificaciones de requerimientos del sistema. Sin embargo, debido a que los requerimientos del sistema son más detallados que los requerimientos del usuario, las especificaciones en lenguaje natural pueden ser confusas y difíciles de entender:

1. La comprensión del lenguaje natural depende de que los lectores y redactores de la especificación utilicen las mismas palabras para el mismo concepto. Esto conduce a malas interpretaciones debido a la ambigüedad del lenguaje natural. Jackson (Jackson, 1995) da un excelente ejemplo de esto cuando analiza las señales mostradas en una escalera mecánica. Éstas indican «Se deben utilizar zapatos» y «Los perros deben cargarse». Se dejan al lector las diferentes interpretaciones de estas frases.
2. Una especificación de requerimientos en lenguaje natural es demasiado flexible. Puede decir lo mismo de formas completamente diferentes. Se deja al lector decidir cuándo los requerimientos son los mismos y cuándo diferentes.
3. No existe una forma fácil de modularizar los requerimientos en lenguaje natural. Puede ser difícil encontrar todos los requerimientos relacionados. Para descubrir la consecuencia de un cambio, puede ser necesario mirar todos los requerimientos en vez de sólo un grupo de requerimientos relacionados.

Debido a estos problemas, las especificaciones de requerimientos redactadas en lenguaje natural son propensas a malas interpretaciones. A menudo éstas no se descubren hasta las fases posteriores del proceso del software, y resolverlas puede resultar muy costoso.

Es esencial redactar los requerimientos del usuario en un lenguaje que los no especialistas puedan entender. Sin embargo, se pueden redactar los requerimientos del sistema en unas notaciones más especializada (Figura 6.11). Éstas incluyen un lenguaje natural estructurado y

Lenguaje natural estructurado	Este enfoque depende de la definición de formularios o plantillas estándares para expresar la especificación de requerimientos.
Lenguajes de descripción de diseño	Este enfoque utiliza un lenguaje similar a uno de programación, pero con características más abstractas, para especificar los requerimientos por medio de la definición de un modelo operativo del sistema. Este enfoque no se utiliza ampliamente en la actualidad, aunque puede ser útil para especificaciones de interfaces.
Notaciones gráficas	Para definir los requerimientos funcionales del sistema, se utiliza un lenguaje gráfico, complementado con anotaciones de texto. Uno de los primeros lenguajes gráficos fue SADT (Ross, 1977) (Schoman y Ros, 1977). Actualmente, se suelen utilizar las descripciones de casos de uso (Jacobsen *et al.*, 1993) y los diagramas de secuencia (Stevens y Pooley, 1999).
Especificaciones matemáticas	Son notaciones que se basan en conceptos matemáticos como el de las máquinas de estado finito o los conjuntos. Estas especificaciones no ambiguas reducen los argumentos sobre la funcionalidad del sistema entre el cliente y el contratista. Sin embargo, la mayoría de los clientes no comprenden las especificaciones formales y son reacios a aceptarlas como un contrato del sistema.

Figura 6.11 Notaciones para la especificación de requerimientos.

estilizado, modelos gráficos de los requerimientos como los casos de uso para las especificaciones matemáticas formales. En este capítulo, se explica cómo el lenguaje natural estructurado complementado con modelos gráficos sencillos puede utilizarse para redactar los requerimientos del sistema. Se estudia el modelado de sistemas gráficos en el Capítulo 8 y la especificación formal del sistema en el Capítulo 10.

6.3.1 Especificaciones en lenguaje estructurado

El lenguaje natural estructurado es una forma de redactar los requerimientos del sistema donde la libertad del redactor de los requerimientos está limitada y donde todos los requerimientos se redactan de una forma estándar. La ventaja de este enfoque es que mantiene la mayor parte de la expresividad y comprensión del lenguaje natural, pero asegura que se imponga cierto grado de uniformidad en la especificación. Las notaciones del lenguaje estructurado limitan la terminología que se puede utilizar y emplean plantillas para especificar los requerimientos del sistema. Pueden incorporar construcciones de control derivadas de los lenguajes de programación y manifestaciones gráficas para dividir la especificación.

Heninger (Heninger, 1980) describe uno de los primeros proyectos que utilizó el lenguaje natural estructurado para especificar los requerimientos del sistema. Se diseñaron formularios de propósito especial para describir la entrada, la salida y las funciones de un sistema software para un avión. Estos requerimientos se especificaron utilizando dichos formularios.

Para utilizar un enfoque basado en formularios para especificar los requerimientos del sistema, se deben definir una o más formularios o plantillas estándares para expresar estos requerimientos. Se puede estructurar la especificación alrededor de los objetos que manipula el sistema, las funciones ejecutadas por el sistema o los eventos procesados por éste. En la Figu-

Función	**Calcular la dosis de insulina: nivel de azúcar en sangre.**
Descripción	**Calcula la dosis de insulina a suministrar cuando el nivel medido actual del azúcar está en la zona segura entre 3 y 7 unidades.**
Entradas	**Lectura del azúcar actual (r2), las dos lecturas previas (r0 y r1).**
Fuente	**Lectura actual del azúcar del sensor. Otras lecturas de la memoria.**
Salidas	**CompDose: la dosis en insulina a suministrar.**
Destino	**Bucle de control principal.**
Acción:	**CompDose es cero si el nivel de azúcar está estable o disminuyendo o si el nivel está aumentando pero la tasa de incremento disminuyendo. Si el nivel está aumentando y la tasa de incremento disminuyendo, CompDose se calcula dividiendo la diferencia entre el nivel de azúcar actual y el nivel anterior por 4 y redondeando el resultado. Si el resultado se redondea a cero, se fija CompDose a la dosis mínima que puede ser suministrada.**
Requerimientos	**Las dos lecturas previas para poder calcular la tasa de cambio del azúcar.**
Precondición	**La reserva de insulina contiene al menos el máximo permitido para una única dosis de insulina.**
Postcondición	**r0 es reemplazada por r1 y r1 es reemplazada por r2.**
Efectos colaterales	**Ninguno.**

Figura 6.12 La especificación de requerimientos del sistema utilizando un formulario estándar.

ra 6.12 se muestra un ejemplo de una especificación basada en formularios. Se ha tomado este ejemplo del sistema de bomba de insulina que se introdujo en el Capítulo 3.

La bomba de insulina basa sus cálculos de las necesidades de insulina del usuario en la tasa de cambio de los niveles de azúcar en la sangre. Estas tasas de cambio se calculan utilizando las lecturas actuales y previas. Puede descargar una versión completa de la especificación para la bomba de insulina de la página web del libro.

Cuando se utiliza un formulario estándar para especificar los requerimientos funcionales, se debe incluir la siguiente información:

1. Descripción de la función o entidad a especificar.
2. Descripción de sus entradas y de dónde provienen.
3. Descripción de sus salidas y hacia dónde van.
4. Indicación de qué otras entidades se utilizan (la parte de *requerimientos*).
5. Si se utiliza un enfoque funcional, una precondición que indique lo que se debe cumplir antes de invocar a la función y una postcondición que especifique lo que será verdad una vez invocada dicha función.
6. Descripción de los efectos colaterales (si existen) de la operación.

El uso de especificaciones formateadas elimina algunos de los problemas de la especificación en lenguaje natural. Se reduce la variabilidad en la especificación y los requerimientos se organizan de forma más efectiva. Sin embargo, es difícil redactar requerimientos de forma no ambigua, especialmente cuando se requieren cálculos complejos. Puede observarse esto en la descripción mostrada en la Figura 6.12, donde no queda claro qué es lo que ocurre si la precondición no es satisfecha.

Para abordar el problema, se puede añadir información adicional a los requerimientos en lenguaje natural utilizando tablas o modelos gráficos del sistema. Éstos pueden mostrar cómo se ejecutan los cálculos, cómo el sistema establece los cambios, cómo los usuarios interactúan con el sistema y cómo se desarrollan las secuencias de acciones.

Las tablas son especialmente útiles cuando hay varias situaciones alternativas posibles y se necesita describir las acciones a tomar para cada una de ellas. La Figura 6.13 es una descripción revisada de los cálculos de la dosis de insulina.

Los modelos gráficos son los más útiles cuando se necesite mostrar cómo cambia el estado (véase el Capítulo 8) o cuando se necesite describir una secuencia de acciones. La Figura 6.14 ilustra la secuencia de acciones cuando un usuario desea retirar dinero de un cajero automático (ATM).

Debe leerse un diagrama de secuencias de arriba abajo para ver el orden de las acciones que tienen lugar. En la Figura 6.14, hay tres subsecuencias básicas:

1. *Validar tarjeta*. Se valida la tarjeta del usuario verificando el número de tarjeta y su número secreto (PIN).

Nivel de azúcar disminuyendo (r2 < r1)	**CompDose = 0**
Nivel de azúcar estable (r2 = r1)	**CompDose = 0**
Nivel de azúcar aumentando y tasa de incremento disminuyendo ((r2 − r1) < (r1 − r0))	**CompDose = 0**
Nivel de azúcar aumentando y tasa de incremento estable o aumentando. ((r2 − r1) > (r1 − r0))	**CompDose = redondear((r2 − r1)/4)** **Si resultado redondeado = 0 entonces CompDose = DosisMinima**

Figura 6.13 Especificación tabular del cálculo.

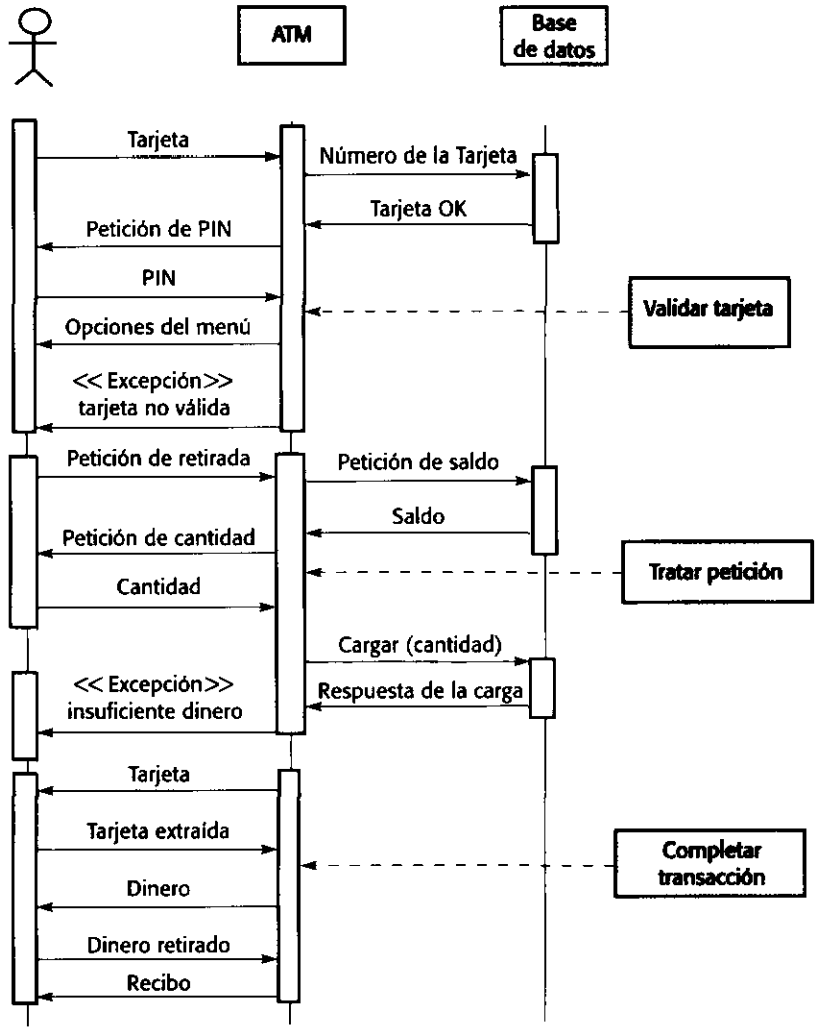


Figura 6.14 Diagrama de secuencia de la retirada de dinero en un ATM.

2. *Tratar petición*. El sistema trata la petición del usuario. Para una retirada de dinero, se debe consultar la base de datos para comprobar el saldo del usuario y cargar la cantidad retirada. Fíjese aquí en la excepción si el solicitante no tiene suficiente dinero en su cuenta.
3. *Completar transacción*. Se devuelve la tarjeta del usuario y, cuando se ha extraído, se entrega el dinero y el recibo.

Se verán de nuevo los diagramas de secuencia en el Capítulo 8, que trata los modelos de sistemas, y en el Capítulo 14, que estudia el diseño orientado a objetos.

6.4 Especificación de la interfaz

Casi todos los sistemas software deben funcionar con otros sistemas que ya están implementados e instalados en el entorno. Si el sistema nuevo y los ya existentes deben trabajar juntos, las interfaces de estos últimos deben especificarse de forma precisa. Estas especificaciones se

deben definir al inicio del proceso y se incluyen (quizás como un apéndice) en el documento de requerimientos.

Existen tres tipos de interfaces que pueden definirse:

1. *Interfaces de procedimientos* en las cuales los programas o subsistemas existentes ofrecen una variedad de servicios a los que se accede invocando a los procedimientos de la interfaz. Estas interfaces a veces se denominan Interfaces de Programación de Aplicaciones (APIs).
2. *Estructuras de datos* que pasan de un subsistema a otro. Los modelos gráficos de datos (descritos en el Capítulo 8) son las mejores notaciones para este tipo de descripción. Si es necesario, se pueden generar automáticamente descripciones de programas en Java o C++ de estas descripciones.
3. *Representaciones de datos* (como el orden de los bits) establecidas para un subsistema existente. Estas interfaces son muy comunes en sistemas de tiempo real embebido. Algunos lenguajes de programación como Ada (aunque no Java) soportan este nivel de especificación. Sin embargo, la mejor forma de describir éstos es probablemente utilizar un diagrama de la estructura con anotaciones que expliquen la función de cada grupo de bits.

Las notaciones formales, analizadas en el Capítulo 10, permiten definir interfaces de una forma no ambigua, pero por su naturaleza no son comprensibles sin una formación especial. Raramente se utilizan en la práctica para la especificación de interfaces, aunque son ideales a estos efectos. Se puede utilizar un lenguaje de programación como Java para describir la sintaxis de la interfaz. Sin embargo, esto se tiene que complementar con una descripción adicional que explique la semántica de cada una de las operaciones definidas.

La Figura 6.15 es un ejemplo de definición de interfaz de procedimiento definida en Java. En este caso, ésta es una interfaz de procedimiento ofrecida por un servidor de impresión. Éste gestiona una cola de espera de las peticiones de impresión de archivos en diferentes impresoras. Los usuarios pueden examinar dicha cola de espera asociada con una impresora y pueden eliminar sus trabajos de impresión de esa cola. También pueden cambiar sus trabajos de una impresora a otra. La especificación en la Figura 6.15 es un modelo abstracto del servidor de impresión que no muestra los detalles de la interfaz. La funcionalidad de las operaciones de ésta se puede definir utilizando el lenguaje natural estructurado o la descripción tabular.

6.5 El documento de requerimientos del software

El documento de requerimientos del software (algunas veces denominado especificación de requerimientos del software o SRS) es la declaración oficial de qué deben implementar los

```
Interface PrintServer {
// define un servidor de impresión abstracto
// requiere: interfaz Printer, interfaz PrintDoc
// proporciona: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter
   void initialize ( Printer p ) ;
   void print ( Printer p, PrintDoc d ) ;
   void displayPrintQueue ( Printer p ) ;
   void cancel PrintJob (printer p, PriDoc d) ;
   void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

Figura 6.15
La descripción en PDL basado en Java de una interfaz del servidor de impresión.

desarrolladores del sistema. Debe incluir tanto los requerimientos del usuario para el sistema como una especificación detallada de los requerimientos del sistema. En algunos casos, los dos tipos de requerimientos se pueden integrar en una única descripción. En otros, los requerimientos del usuario se definen en una introducción a la especificación de los requerimientos del sistema. Si existe un gran número de requerimientos, los detalles de los requerimientos del sistema se pueden presentar en un documento separado.

El documento de requerimientos tiene un conjunto diverso de usuarios que va desde los altos cargos de la organización que pagan por el sistema, hasta los ingenieros responsables de desarrollar el software. La Figura 6.16, tomada de mi libro con Gerald Kotonya sobre ingeniería de requerimientos (Kontonya y Sommerville, 1998), ilustra los posibles usuarios del documento y cómo lo utilizan.

La diversidad de posibles usuarios significa que el documento de requerimientos tiene que presentar un equilibrio entre la comunicación de los requerimientos a los clientes, la definición de los requerimientos en el detalle exacto para los desarrolladores y probadores, y la inclusión de información sobre la posible evolución del sistema. La información sobre cambios previstos puede ayudar a los diseñadores del sistema a evitar decisiones de diseño restrictivas y a los ingenieros encargados del mantenimiento del sistema, quienes tienen que adaptar el sistema a los nuevos requerimientos.

El nivel de detalle que se debe incluir en un documento de requerimientos depende del tipo de sistema que se desarrolle y del proceso de desarrollo utilizado. Cuando el sistema se desarrolle por un contratista externo, las especificaciones de los sistemas críticos necesitan ser claras y muy detalladas. Cuando haya más flexibilidad en los requerimientos y cuando se utilice un proceso de desarrollo iterativo dentro de la empresa, el documento de

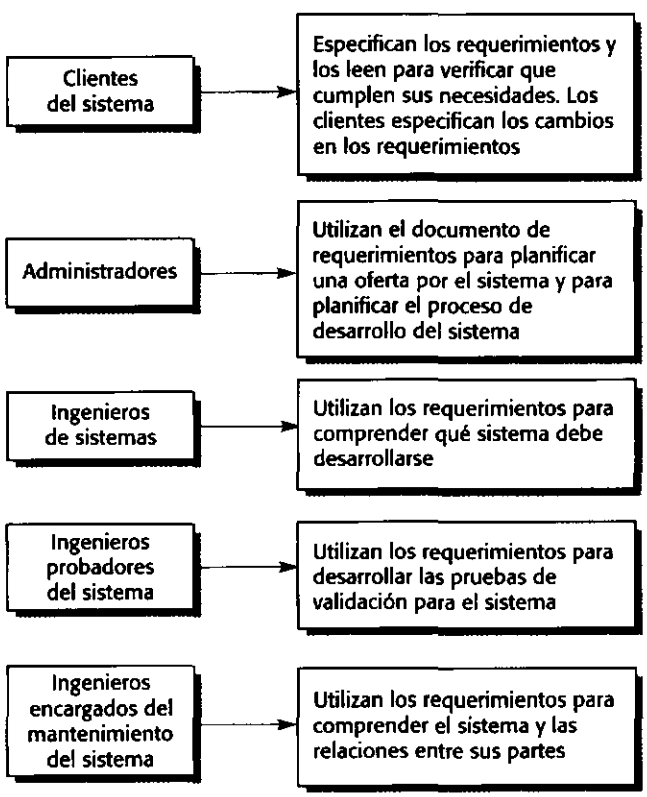


Figura 6.16 Usuarios de un documento de requerimientos.

requerimientos puede ser mucho menos detallado y cualquier ambigüedad resuelta durante el desarrollo del sistema.

Varias organizaciones grandes, como el Departamento de Defensa de los Estados Unidos y el IEEE, han definido estándares para los documentos de requerimientos. Davis (Davis, 1993) analiza algunos de estos estándares y compara sus contenidos. El estándar más ampliamente conocido es el IEEE/ANSI 830-1998 (IEEE, 1998). Este estándar IEEE sugiere la siguiente estructura para los documentos de requerimientos:

1. **Introducción**
   - 1.1 Propósito del documento de requerimientos
   - 1.2 Alcance del producto
   - 1.3 Definiciones, acrónicos y abreviaturas
   - 1.4 Referencias
   - 1.5 Descripción del resto del documento
2. **Descripción general**
   - 2.1 Perspectiva del producto
   - 2.2 Funciones del producto
   - 2.3 Características del usuario
   - 2.4 Restricciones generales
   - 2.5 Suposiciones y dependencias
3. **Requerimientos específicos:** incluyen los requerimientos funcionales, no funcionales y de interfaz. Obviamente, ésta es la parte más sustancial del documento, pero debido a la amplia variabilidad en la práctica organizacional, no es apropiado definir una estructura estándar para esta sección. Los requerimientos pueden documentar las interfaces externas, describir la funcionalidad y el rendimiento del sistema, especificar los requerimientos lógicos de la base de datos, las restricciones de diseño, las propiedades emergentes del sistema y las características de calidad.
4. **Apéndices**
5. **Índice**

Aunque el estándar IEEE no es ideal, contiene muchos consejos sobre cómo redactar los requerimientos y cómo evitar problemas. Es muy general para que pueda ser un estándar de una organización. Es un marco general que se puede transformar y adaptar para definir un estándar ajustado a las necesidades de una organización en particular. La Figura 6.17 ilustra una posible organización para un documento de requerimientos que se basa en el estándar IEEE. Sin embargo, se ha ampliado para incluir información sobre la evolución predicha del sistema. Esto fue propuesto por primera vez por Heninger (Heninger, 1980) y, como se ha indicado, ayuda a las personas encargadas del mantenimiento del sistema y puede permitir a los diseñadores incluir soporte para futuras características del sistema.

Por supuesto, la información que se incluya en un documento de requerimientos debe depender del tipo de software a desarrollar y del enfoque de desarrollo que se utilice. Si se adopta un enfoque evolutivo para un producto de software (por ejemplo), el documento de requerimientos dejará fuera muchos de los capítulos detallados sugeridos anteriormente. El interés estará en definir los requerimientos del usuario y los requerimientos del sistema no funcionales de alto nivel. En este caso, los diseñadores y programadores utilizan su juicio para decidir cómo satisfacer el esquema de los requerimientos del usuario para el sistema.

Por el contrario, cuando el software es parte de un proyecto de ingeniería de sistemas grande que incluye la interacción de sistemas hardware y software, a menudo es fundamental definir con mucho detalle los requerimientos. Esto significa que el documento de requerimien-

tos probablemente sea muy extenso y deba incluir la mayoría si no la totalidad de los capítulos que se muestran en la Figura 6.17. Para documentos extensos, es de particular importancia incluir una tabla de contenido comprensible y un índice del documento para que así los lectores puedan encontrar la información que necesitan.

El documento de requerimientos es fundamental cuando un contratista exterior está desarrollando el sistema software. Sin embargo, los métodos de desarrollo ágiles sostienen que los requerimientos cambian tan rápidamente que un documento de requerimientos se queda desfasado en cuanto se redacta, por lo que el esfuerzo en gran parte se malgasta. Más que un documento formal, enfoques como la programación extrema (Beck, 1999) proponen que los requerimientos del usuario deberían ser recogidos incrementalmente y escritos en tarjetas. El usuario entonces da prioridad a los requerimientos que se han de implementar en el siguiente incremento del sistema.

Para sistemas de negocio donde los requerimientos son inestables, pienso que este enfoque es bueno. Sin embargo, argumentaría que todavía es útil redactar un breve documento de soporte que defina el negocio y los requerimientos de confiabilidad del sistema. Es fácil olvidarse de los requerimientos que se aplican al sistema en su totalidad al centrarse en los requerimientos funcionales para la siguiente entrega del sistema.

Prefacio	**Debe definir los posibles lectores del documento y describir su versión de la historia, incluyendo un fundamento para la creación de una nueva versión y un resumen de los cambios hechos en cada una.**
Introducción	**Debe describir la necesidad del sistema. Debe describir brevemente sus funciones y explicar cómo trabajará con otros sistemas. Debe describir la manera en que éste se adhiere al negocio total u objetivos estratégicos de la organización que solicita el software.**
Glosario	**Debe definir los términos técnicos utilizados en el documento. No se deben hacer suposiciones de la experiencia o pericia del lector.**
Definición de requerimientos del usuario	**En esta sección se deben describir los servicios que se proporcionan al usuario y los requerimientos no funcionales del sistema. Esta descripción puede utilizar lenguaje natural, diagramas u otras notaciones que sea comprensibles para los clientes. Se deben especificar los estándares de productos y procesos a seguir.**
Arquitectura del sistema	**Este capítulo debe presentar una visión general de alto nivel de la arquitectura prevista del sistema que muestre la distribución de funciones en los módulos del sistema. Se deben resaltar los componentes arquitectónicos reutilizados.**
Especificación de requerimientos del sistema	**Debe describir con mayor detalle los requerimientos funcionales y no funcionales. Si es necesario, se pueden enfatizar los no funcionales; por ejemplo, se pueden definir las interfaces con otros sistemas.**
Modelos del sistema	**Se deben exponer uno o más modelos del sistema que muestren las relaciones entre los componentes del sistema y el sistema y su entorno. Éstos podrían ser modelos de objetos, modelos de flujos de datos y modelos de datos semánticos.**
Evolución del sistema	**Debe describir las suposiciones fundamentales sobre las cuales se basa el sistema y los cambios previstos debido a la evolución del hardware, cambios en las necesidades del usuario, etcétera.**
Apéndices	**Debe proporcionar información detallada y precisa relacionada con la aplicación que se desarrolla. Algunos ejemplos de apéndices que pueden incluirse son las descripciones del hardware y de la base de datos. Los requerimientos de hardware definen las configuraciones mínima y óptima del sistema. Los de la base de datos definen la organización lógica de los datos utilizados por el sistema y las relaciones entre los datos.**
Índice	**Se pueden incluir varios índices en el documento. Además de un índice alfabético, puede haber un índice de diagramas, un índice de funciones, etcétera.**

Figura 6.17 La estructura de un documento de requerimientos.

PUNTOS CLAVE

- Los requerimientos para un sistema software determinan lo que debe hacer el sistema y definen las restricciones en su funcionamiento e implementación.
- Los requerimientos funcionales son declaraciones de los servicios que el sistema debe proporcionar o son descripciones de cómo se deben llevar a cabo algunos cálculos. Los requerimientos del dominio son requerimientos funcionales que se derivan de las características del dominio de aplicación.
- Los requerimientos no funcionales restringen el sistema en desarrollo y el proceso de desarrollo que se debe utilizar. Pueden ser requerimientos del producto, organizacionales o externos. A menudo están relacionados con las propiedades emergentes del sistema y, por lo tanto, se aplican al sistema completo.
- Los requerimientos del usuario son para el uso por la gente relacionada con la utilización y obtención del sistema. Se deben redactar en lenguaje natural, con tablas y diagramas que sean fáciles de entender.
- Los requerimientos del sistema se utilizan para comunicar, de forma precisa, las funciones que debe proporcionar el sistema. Para reducir la ambigüedad, se pueden redactar en un formulario estructurado del lenguaje natural complementado con tablas y modelos del sistema.
- Los documentos de requerimientos de software son la declaración acordada de los requerimientos del sistema. Se deben organizar de tal forma que puedan ser utilizados tanto por los clientes del sistema como por los desarrolladores del software.
- El estándar IEEE para los documentos de requerimientos es un punto de partida útil para estándares más específicos de especificación de requerimientos.

LECTURAS ADICIONALES

Software Requirements, 2nd ed. Este libro, diseñado por los escritores y usuarios de los requerimientos, describe las buenas prácticas de la ingeniería de requerimientos. (K. M. Weigers, 2003, Microsoft Press.)

Mastering the Requirements Process. Un libro bien escribo y fácil de leer que está basado en un método en concreto (VOLERE), pero el cual también incluye muchos buenos consejos generales sobre la ingeniería de requerimientos. (S. Robertson y J. Robertson, 1999, Addison-Wesley.)

Requirements Engineering: Processes and Techniques. Este libro abarca todos los aspectos del proceso de ingeniería de requerimientos y trata técnicas concretas de la especificación de requerimientos. (G. Kotonya e I. Sommerville, 1998, John Wiley & Sons.)

Software Requirements Engineering. Esta colección de artículos sobre ingeniería de requerimientos incluye varios artículos relevantes como «Recommended Practice for Software Requirements Specification», un estudio del estándar IEEE para los documentos de requerimientos. [R. H. Thayer y M. Dorfman (eds.), 1997, IEEE Computer Society Press.]

EJERCICIOS

6.1 Identifique y comente brevemente cuatro tipos de requerimientos que se pueden definir para un sistema informático.

6.2 Comente los problemas de la utilización del lenguaje natural para definir los requerimientos del usuario y del sistema, y muestre, utilizando pequeños ejemplos, cómo el estructurar el lenguaje natural en formularios puede ayudar a evitar algunas de estas dificultades.

6.3 Descubra las ambigüedades u omisiones en la siguiente declaración de requerimientos de una parte de un sistema expendedor de billetes.

Un sistema automático de expedición de billetes vende billetes de tren. Los usuarios seleccionan su destino e introducen una tarjeta de crédito y un número de identificación personal. El billete de tren se expide y se carga su cuenta de la tarjeta de crédito. Cuando el usuario presiona el botón de inicio, se activa un menú que muestra los posibles destinos, junto con un mensaje para el usuario que le indica que seleccione el destino. Una vez que se ha seleccionado un destino, se pide a los usuarios que introduzcan su tarjeta de crédito. Se comprueba su validez y entonces se le pide introducir un identificador personal. Cuando la transacción de crédito se haya validado, se expide el billete.

6.4 Vuelva a redactar la descripción anterior utilizando el enfoque estructurado descrito en este capítulo. Resuelva de forma apropiada las ambigüedades identificadas.

6.5 Dibuje un diagrama de secuencias que muestre las acciones llevadas a cabo en el sistema expendedor de billetes. Puede hacer algunas suposiciones razonables sobre el sistema. Ponga especial atención en la especificación de los errores del usuario.

6.6 Utilizando la técnica sugerida aquí, en la que el lenguaje natural se presenta en una forma estándar, redacte requerimientos del usuario verosímiles para las siguientes funciones:

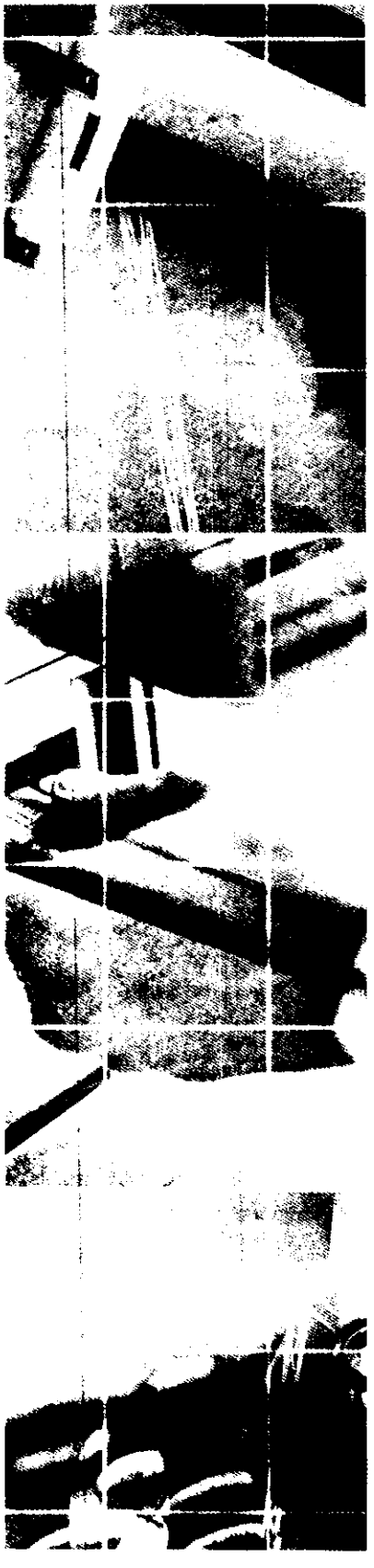
- La función de expedición de dinero en un cajero automático de un banco.
- La verificación de ortografía y la función de corrección en un procesador de texto.
- Un sistema de autoservicio de bombas de gasolina que incluye un lector de tarjetas de crédito. El cliente pasa la tarjeta a través del lector y especifica la cantidad de combustible requerido. Éste se entrega y se hace el cargo a la cuenta del cliente.

6.7 Describa cuatro tipos de requerimientos no funcionales que pueden existir en un sistema. Dé ejemplos de cada uno de estos tipos de requerimientos.

6.8 Redacte un conjunto de requerimientos no funcionales para el sistema expendedor de billetes, especificando su fiabilidad y su respuesta en el tiempo.

6.9 Sugiera la manera en que un ingeniero responsable de preparar la especificación de requerimientos del sistema podría controlar las relaciones entre los requerimientos funcionales y no funcionales.

6.10 Ha obtenido un trabajo con un usuario de software quien ha contratado a su anterior compañía para desarrollar un sistema. Usted descubre que la interpretación de su compañía actual de los requerimientos es diferente de la tomada por su anterior compañía. Comente qué haría en tal situación. Usted sabe que los costes de su compañía actual se incrementarán si las ambigüedades no se resuelven. También tiene una responsabilidad de confidencialidad para su anterior compañía.



7

Procesos de la ingeniería de requerimientos

Objetivos

El objetivo de este capítulo es tratar las actividades implicadas en el proceso de ingeniería de requerimientos. Cuando haya leído este capítulo:

- comprenderá las principales actividades de la ingeniería de requerimientos y sus relaciones;
- habrá sido introducido en diversas técnicas para la obtención y análisis de requerimientos;
- comprenderá la importancia de la validación de requerimientos y cómo se utilizan las revisiones de éstos en este proceso;
- comprenderá por qué es necesaria la gestión de requerimientos y cómo ayuda a otras actividades de la ingeniería de requerimientos.

Contenidos

La meta del proceso de ingeniería de requerimientos es crear y mantener un documento de requerimientos del sistema. El proceso general corresponde cuatro subprocesos de alto nivel de la ingeniería de requerimientos. Éstos tratan de la evaluación de si el sistema es útil para el negocio (estudio de viabilidad); el descubrimiento de requerimientos (obtención y análisis); la transformación de estos requerimientos en formularios estándar (especificación), y la verificación de que los requerimientos realmente definen el sistema que quiere el cliente (validación). La Figura 7.1 ilustra la relación entre estas actividades. También muestra el documento que se elabora en cada etapa del proceso de ingeniería de requerimientos. La especificación y la documentación se estudian en el Capítulo 6; este capítulo se centra en las actividades de la ingeniería de requerimientos.

Las actividades que se muestran en la Figura 7.1 se refieren al descubrimiento, documentación y verificación de requerimientos. Sin embargo, en casi todos los sistemas los requerimientos cambian. Las personas involucradas desarrollan una mejor comprensión de lo que quieren que haga el software; la organización que compra el sistema cambia; se hacen modificaciones a los sistemas hardware, software y al entorno organizacional. El proceso de gestionar estos cambios en los requerimientos se denomina gestión de requerimientos, tema que se aborda en la sección final de este capítulo.

Se presenta una perspectiva alternativa sobre el proceso de ingeniería de requerimientos en la Figura 7.2. Ésta muestra el proceso como una actividad de tres etapas donde las actividades se organizan como un proceso iterativo alrededor de una espiral. La cantidad de dinero y esfuerzo dedicados a cada actividad en una iteración depende de la etapa del proceso general y del tipo de sistema desarrollado. Al principio del proceso, se dedicará la mayor parte del esfuerzo a la comprensión del negocio de alto nivel y los requerimientos no funcionales y del usuario. Al final del proceso, en el anillo exterior de la espiral, se dedicará un mayor esfuerzo a la ingeniería de requerimientos del sistema y al modelado de éste.

Este modelo en espiral satisface enfoques de desarrollo en los cuales los requerimientos se desarrollan a diferentes niveles de detalle. El número de iteraciones alrededor de la espiral puede variar, por lo que se puede salir de la espiral después de que se hayan obtenido algunos o todos los requerimientos del usuario. Si la actividad de construcción de prototipos mostrada debajo de la validación de requerimientos se extiende para incluir el desarrollo iterativo, como se indica en el Capítulo 17, este modelo permite que los requerimientos y la implementación del sistema se desarrollen al mismo tiempo.

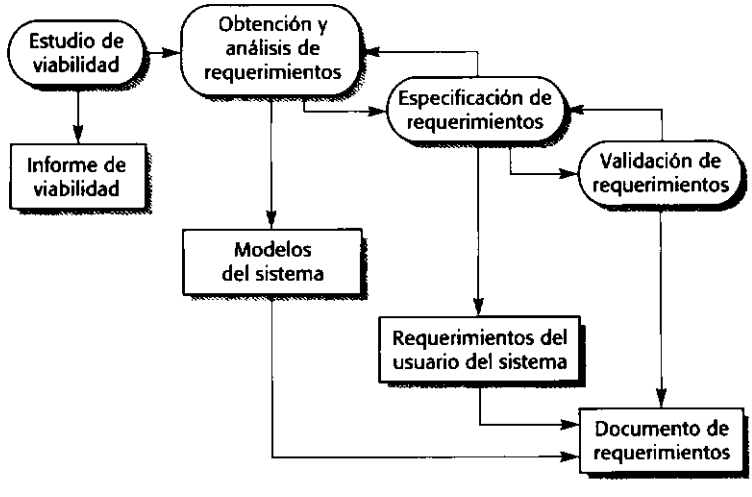


Figura 7.1
El proceso de ingeniería de requerimientos.

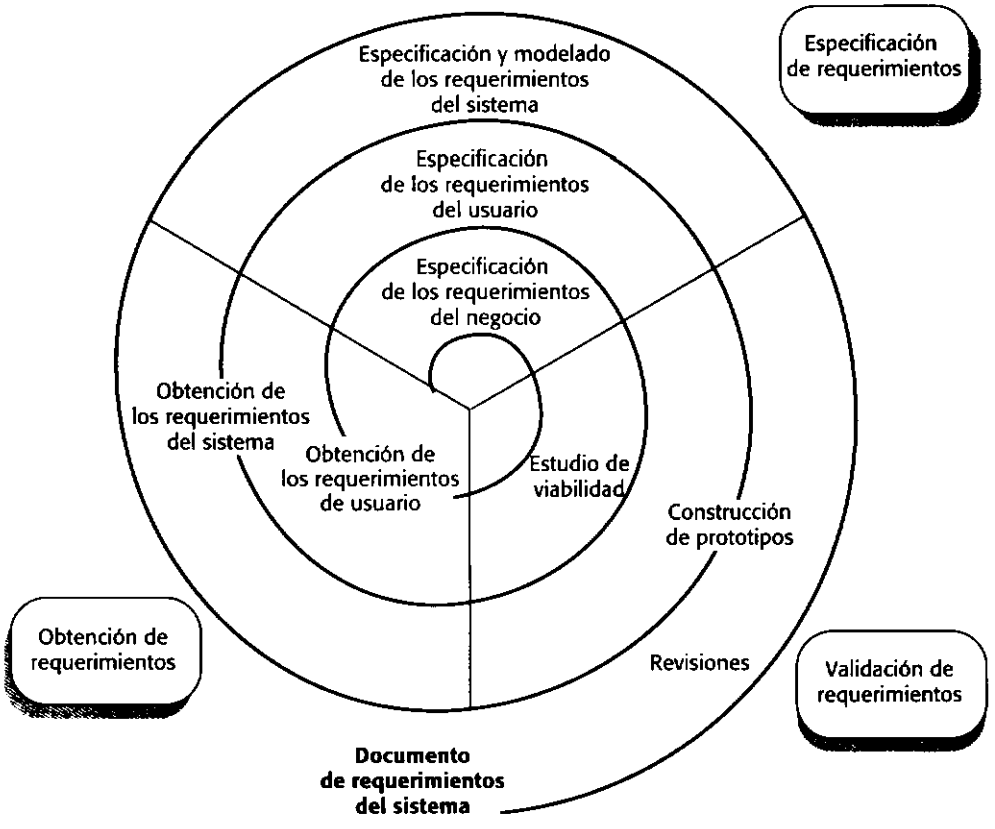


Figura 7.2 Modelo en espiral de los procesos de la ingeniería de requerimientos.

Algunas personas consideran a la ingeniería de requerimientos como el proceso de aplicar un método de análisis estructurado, como el análisis orientado a objetos (Larman, 2002). Éste comprende analizar el sistema y desarrollar un conjunto de modelos gráficos del mismo, como los modelos de casos de uso, que sirven como una especificación del sistema. El conjunto de modelos describe el comportamiento del sistema al cual se le agregan notas con información adicional que detallan, por ejemplo, el rendimiento o fiabilidad requeridos.

Aunque los métodos estructurados juegan un cierto papel en el proceso de ingeniería de requerimientos, existe mucho más en dicha ingeniería de lo que se abarca en estos métodos. La obtención de requerimientos, en particular, es una actividad centrada en las personas, y a éstas no les gustan las restricciones impuestas por modelos de sistemas rígidos. Aquí se tratan los enfoques generales de la ingeniería de requerimientos, y en el Capítulo 8 se examinan los métodos estructurados y los modelos del sistema.

7.1 Estudios de viabilidad

Para todos los sistemas nuevos, el proceso de ingeniería de requerimientos debería empezar con un estudio de viabilidad. La entrada de éste es un conjunto de requerimientos de negocio preliminares, una descripción resumida del sistema y de cómo éste pretende contribuir a los procesos del negocio. Los resultados del estudio de viabilidad deberían ser un informe que recomiende si merece o no la pena seguir con la ingeniería de requerimientos y el proceso de desarrollo del sistema.

Un estudio de viabilidad es un estudio corto y orientado a resolver varias cuestiones:

1. ¿Contribuye el sistema a los objetivos generales de la organización?
2. ¿Se puede implementar el sistema utilizando la tecnología actual y dentro de las restricciones de coste y tiempo?
3. ¿Puede integrarse el sistema con otros sistemas existentes en la organización?

La cuestión de si el sistema contribuye a los objetivos del negocio es crítica. Si no contribuye a estos objetivos, entonces no tiene un valor real en el negocio. Aunque esto pueda parecer obvio, muchas organizaciones desarrollan sistemas que no contribuyen a sus objetivos porque no tienen una clara declaración de estos objetivos, porque no consiguen definir los requerimientos del negocio para el sistema o porque otros factores políticos u organizacionales influyen en la creación del sistema. Aunque no se trata explícitamente, un estudio de viabilidad debería ser parte de la fase de Inicio del Proceso Unificado de Rational, como se comentó en el Capítulo 4.

Llevar a cabo un estudio de viabilidad comprende la evaluación y recopilación de la información, y la redacción de informes. La fase de evaluación de la información identifica la información requerida para contestar las tres preguntas anteriores. Una vez que dicha información se ha identificado, se debería hablar con las fuentes de información para descubrir las respuestas a estas preguntas. Algunos ejemplos de preguntas posibles son:

1. ¿Cómo se las arreglaría la organización si no se implementara este sistema?
2. ¿Cuáles son los problemas con los procesos actuales y cómo ayudaría un sistema nuevo a aliviarlos?
3. ¿Cuál es la contribución directa que hará el sistema a los objetivos y requerimientos del negocio?
4. ¿La información se puede obtener y transferir a otros sistemas de la organización?
5. ¿Requiere el sistema tecnología que no se ha utilizado previamente en la organización?
6. ¿A qué debe ayudar el sistema y a qué no necesita ayudar?

En un estudio de viabilidad, se pueden consultar las fuentes de información, como los jefes de los departamentos donde se utilizará el sistema, los ingenieros de software que están familiarizados con el tipo de sistema propuesto, los expertos en tecnología y los usuarios finales del sistema. Normalmente, se debería intentar completar un estudio de viabilidad en dos o tres semanas.

Una vez que se tiene la información, se redacta el informe del estudio de viabilidad. Debería hacerse una recomendación sobre si debe continuar o no el desarrollo del sistema. En el informe, se pueden proponer cambios en el alcance, el presupuesto y la confección de agendas del sistema y sugerir requerimientos adicionales de alto nivel para éste.

7.2 Obtención y análisis de requerimientos

La siguiente etapa del proceso de ingeniería de requerimientos es la obtención y análisis de requerimientos. En esta actividad, los ingenieros de software trabajan con los clientes y los usuarios finales del sistema para determinar el dominio de la aplicación, qué servicios debe proporcionar el sistema, el rendimiento requerido del sistema, las restricciones hardware, etcétera.

La obtención y análisis de requerimientos pueden afectar a varias personas de la organización. El término *stakeholder* se utiliza para referirse a cualquier persona o grupo que se verá afectado por el sistema, directa o indirectamente. Entre los stakeholders se encuentran los usuarios finales que interactúan con el sistema y todos aquellos en la organización que se pueden ver afectados por su instalación. Otros stakeholders del sistema pueden ser los ingenieros que desarrollan o dan mantenimiento a otros sistemas relacionados, los gerentes del negocio, los expertos en el dominio del sistema y los representantes de los trabajadores.

Obtener y comprender los requerimientos de los stakeholders es difícil por varias razones:

1. Los stakeholders a menudo no conocen lo que desean obtener del sistema informático excepto en términos muy generales; puede resultarles difícil expresar lo que quieren que haga el sistema o pueden hacer demandas irreales debido a que no conocen el coste de sus peticiones.
2. Los stakeholders expresan los requerimientos con sus propios términos de forma natural y con un conocimiento implícito de su propio trabajo. Los ingenieros de requerimientos, sin experiencia en el dominio del cliente, deben comprender estos requerimientos.
3. Diferentes stakeholders tienen requerimientos distintos, que pueden expresar de varias formas. Los ingenieros de requerimientos tienen que considerar todas las fuentes potenciales de requerimientos y descubrir las concordancias y los conflictos.
4. Los factores políticos pueden influir en los requerimientos del sistema. Por ejemplo, los directivos pueden solicitar requerimientos específicos del sistema que incrementarán su influencia en la organización.
5. El entorno económico y de negocios en el que se lleva a cabo el análisis es dinámico. Inevitablemente, cambia durante el proceso de análisis. Por lo tanto, la importancia de ciertos requerimientos puede cambiar. Pueden emerger nuevos requerimientos de nuevos stakeholders que no habían sido consultados previamente.

En la Figura 7.3 se muestra un modelo muy general del proceso de obtención y análisis. Cada organización tendrá su propia versión o instancia de este modelo general, dependiendo de los factores locales como la habilidad del personal, el tipo de sistema a desarrollar y los estándares utilizados. De nuevo, se puede pensar en estas actividades dentro de una espiral de forma que las actividades se entrelazan a medida que el proceso avanza desde el anillo interior a los exteriores de la espiral.

Las actividades del proceso son:

1. *Descubrimiento de requerimientos.* Es el proceso de interactuar con los stakeholders del sistema para recopilar sus requerimientos. Los requerimientos del dominio de los stakeholders y la documentación también se descubren durante esta actividad.
2. *Clasificación y organización de requerimientos.* Esta actividad toma la recopilación no estructurada de requerimientos, grupos relacionados de requerimientos y los organiza en grupos coherentes.
3. *Ordenación por prioridades y negociación de requerimientos.* Inevitablemente, cuando existen muchos stakeholders involucrados, los requerimientos entrarán en conflicto. Esta actividad se refiere a ordenar según las prioridades los requerimientos, y a encontrar y resolver los requerimientos en conflicto a través de la negociación.
4. *Documentación de requerimientos.* Se documentan los requerimientos y se entra en la siguiente vuelta de la espiral. Se pueden producir documentos de requerimientos formales o informales.

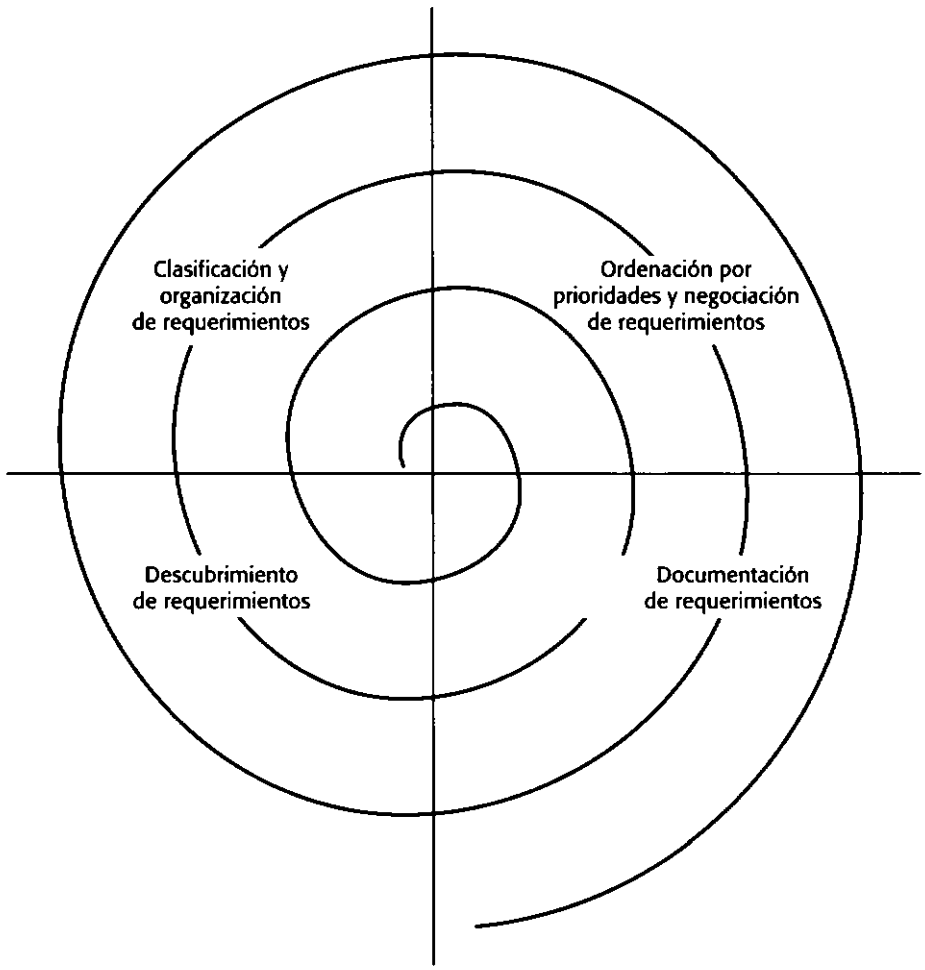


Figura 7.3
El proceso de obtención y análisis de requerimientos.

La Figura 7.3 muestra que la obtención y análisis de requerimientos es un proceso iterativo con retroalimentación continua de cada actividad a las otras actividades. El ciclo del proceso comienza con el descubrimiento de requerimientos y termina con la documentación de requerimientos. La comprensión de los requerimientos por parte del analista mejora con cada vuelta del ciclo.

En este capítulo, se trata principalmente el descubrimiento de requerimientos y en varias técnicas que se han desarrollado para darle soporte. La clasificación y organización de requerimientos principalmente se refiere a la identificación de requerimientos coincidentes de diferentes stakeholders y a la agrupación de requerimientos relacionados. La forma más común de agrupar requerimientos es utilizar un modelo de la arquitectura del sistema para identificar los subsistemas y asociar los requerimientos con cada subsistema. Esto pone de manifiesto que la ingeniería de requerimientos y el diseño arquitectónico no siempre se pueden separar.

Inevitablemente, los stakeholders tienen opiniones diferentes sobre la importancia y prioridad de los requerimientos, y algunas veces estas opiniones están reñidas. Durante el proceso, se deberían organizar frecuentes negociaciones con los stakeholders para que se pueda llegar a acuerdos. Es imposible satisfacer completamente a todos los stakeholders, pero si algún stakeholder piensa que sus opiniones no se han considerado adecuadamente, deliberadamente puede intentar socavar el proceso de ingeniería de requerimientos.

En la etapa de la documentación de requerimientos, los requerimientos que se han obtenido se documentan de tal forma que se pueden utilizar para ayudar al descubrimiento de nuevos requerimientos. En esta etapa, se puede producir una versión inicial del documento de requerimientos del sistema, pero faltarán secciones y habrá requerimientos incompletos. Por otra parte, los requerimientos se pueden documentar como tablas en un documento o en tarjetas. Redactar requerimientos en tarjetas (el enfoque utilizado en la programación extrema) puede ser muy efectivo, ya que son fáciles de manejar, cambiar y organizar para los stakeholders.

7.2.1 Descubrimiento de requerimientos

El descubrimiento de requerimientos es el proceso de recoger información sobre el sistema propuesto y los existentes y extraer los requerimientos del usuario y del sistema de esta información. Las fuentes de información durante la fase del descubrimiento de requerimientos incluyen la documentación, los stakeholders del sistema y la especificación de sistemas similares. Usted se relaciona con los stakeholders a través de entrevistas y de la observación, y puede utilizar escenarios y prototipos para ayudar al descubrimiento de requerimientos. En esta sección, se analiza un enfoque que ayuda a asegurar a que consiga una amplia cobertura de stakeholders al descubrir requerimientos, y se describen técnicas de descubrimiento de requerimientos entre las que se incluyen entrevistas, escenarios y etnografía. Otras técnicas de descubrimiento de requerimientos que se pueden utilizar son los métodos de análisis estructurado, estudiados en el Capítulo 8, y la construcción de prototipos del sistema, que se trata en el Capítulo 17.

Son stakeholders desde los usuarios finales del sistema hasta los gerentes y stakeholders externos como los reguladores quienes certifican la aceptabilidad del sistema. Por ejemplo, entre los stakeholders del sistema para un cajero automático (ATM) de un banco se encuentran:

1. Los clientes actuales del banco quienes reciben los servicios del sistema
2. Los representantes de otros bancos quienes tienen acuerdos recíprocos que les permiten utilizar otros ATMs
3. Los directores de las sucursales bancarias quienes obtienen información del sistema
4. El personal de ventanilla de las sucursales bancarias quienes están relacionados con el funcionamiento diario del sistema
5. Los administradores de la base de datos quienes son responsables de integrar el sistema con la base de datos de clientes del banco
6. Los administradores de seguridad del banco quienes deben asegurar que el sistema no suponga un riesgo de seguridad
7. Las personas del departamento de marketing del banco quienes probablemente están interesadas en utilizar el sistema como un medio para promocionar al banco
8. Los ingenieros de mantenimiento de hardware y software quienes son responsables de mantener y actualizar el hardware y el software
9. Los reguladores de la banca nacional quienes son responsables de asegurar que el sistema se ajusta a las regulaciones de la banca

Además de los stakeholders del sistema, ya hemos visto que los requerimientos pueden venir del dominio de la aplicación y de otros sistemas que interactúan con el sistema a especificar. Todos éstos se deben considerar durante el proceso de obtención de requerimientos.

Estas fuentes de requerimientos (stakeholders, dominio, sistemas) se pueden representar como puntos de vista del sistema, donde cada uno presenta un subconjunto de requerimientos para el sistema. Cada punto de vista proporciona una perspectiva nueva en el sistema, pero éstas no son completamente independientes. Por lo general coinciden parcialmente, por lo que tienen requerimientos comunes.

Puntos de vista

Los enfoques orientados a puntos de vista para la ingeniería de requerimientos (Mullery, 1979; Finkelstein *et al.*, 1992; Kotonya y Sommerville, 1992; Kotonya y Sommerville, 1996) organizan tanto el proceso de obtención como los requerimientos mismos utilizando puntos de vista. Un punto clave del análisis orientado a puntos de vista es que reconoce varias perspectivas y proporciona un marco de trabajo para descubrir conflictos en los requerimientos propuestos por diferentes stakeholders.

Los puntos de vista se pueden utilizar como una forma de clasificar los stakeholders y otras fuentes de requerimientos. Existen tres tipos genéricos de puntos de vista:

1. *Puntos de vista de los interactuadores:* representan a las personas u otros sistemas que interactúan directamente con el sistema. En el sistema del ATM del banco, son ejemplos de estos puntos de vista los clientes del banco y la base de datos de las cuentas bancarias.
2. *Puntos de vista indirectos:* representan a los stakeholders que no utilizan el sistema ellos mismos pero que influyen en los requerimientos de algún modo. En el sistema del ATM del banco, son ejemplos de puntos de vista indirectos la gerencia del banco y el personal de seguridad.
3. *Puntos de vista del dominio:* representan las características y restricciones del dominio que influyen en los requerimientos del sistema. En el sistema del ATM del banco, un ejemplo de un punto de vista del dominio serían los estándares que se han desarrollado para las comunicaciones interbancarias.

Por lo general, estos puntos de vista proporcionan diferentes tipos de requerimientos. Los puntos de vista de los interactuadores proporcionan requerimientos detallados del sistema que cubren las características e interfaces del mismo. Los puntos de vista indirectos es más probable que proporcionen requerimientos y restricciones organizacionales de alto nivel. Los puntos de vista del dominio proporcionan restricciones del dominio que se aplican al sistema.

La identificación inicial de los puntos de vista que son relevantes a un sistema a veces puede ser difícil. Para ayudar a este proceso, se debería intentar identificar tipos de puntos de vista más específicos:

1. Los proveedores de servicios al sistema y los receptores de los servicios del sistema.
2. Los sistemas que deben interactuar directamente con el sistema a especificar.
3. Las regulaciones y estándares que se aplican al sistema.
4. Las fuentes de los requerimientos no funcionales y de negocio del sistema.
5. Los puntos de vista de la ingeniería que reflejan los requerimientos de las personas que tienen que desarrollar, administrar y mantener el sistema.
6. Los puntos de vista del marketing y otros que generan requerimientos sobre las características del producto esperadas por los clientes y cómo el sistema debería reflejar la imagen externa de la organización.

Casi todos los sistemas organizacionales deben interoperar con otros sistemas en la organización. Cuando se diseña un sistema nuevo, se deben diseñar las interacciones con los otros sistemas. Las interfaces ofrecidas por estos otros sistemas ya se han diseñado. Éstas pueden poner requerimientos y restricciones en el sistema nuevo. Además, los sistemas nuevos se pueden tener que ajustar a las regulaciones y estándares existentes, y éstos restringen los requerimientos del sistema.

Como se ha indicado anteriormente en el capítulo, se deben identificar los requerimientos no funcionales y de negocio de alto nivel al inicio del proceso de ingeniería de requerimientos. Las fuentes de estos requerimientos pueden ser puntos de vista útiles en un proceso más detallado. Pueden ser capaces de ampliar y desarrollar los requerimientos de alto nivel en requerimientos del sistema más específicos.

Los puntos de vista de la ingeniería pueden ser importantes por dos razones. En primer lugar, los ingenieros que desarrollan el sistema pueden tener experiencia con sistemas similares y sugerir requerimientos a partir de esa experiencia. En segundo lugar, el personal técnico que tiene que administrar y mantener el sistema puede tener requerimientos que ayuden a simplificar el soporte del sistema. Los requerimientos de administración del sistema son cada vez más importantes debido a que los costes de administración del sistema son una proporción creciente de los costes totales para un sistema.

Finalmente, los puntos de vista que proporcionan requerimientos pueden venir de los departamentos de marketing y asuntos externos en una organización. Esto es especialmente cierto para sistemas basados en web, particularmente sistemas de comercio electrónico y productos software empaquetados. Los sistemas basados en web deben presentar una imagen favorable de la organización además de entregar funcionalidad al usuario. Para productos software, el departamento de marketing debería conocer qué características del sistema lo harán más comercializable para los compradores potenciales.

Para cualquier sistema no trivial, existe un enorme número de posibles puntos de vista, y es prácticamente imposible obtener requerimientos de todos ellos. Por lo tanto, es importante organizar y estructurar los puntos de vista en una jerarquía. Es probable que los puntos de vista en la misma rama compartan requerimientos comunes.

Como ilustración, considere la jerarquía de puntos de vista mostrada en la Figura 7.4. Es un diagrama relativamente sencillo de los puntos de vista que se pueden consultar en la obtención de requerimientos para el sistema LIBSYS. Puede verse que la clasificación de los puntos de vista de los interactuadores, indirectos y de dominio ayuda a identificar las fuentes de los requerimientos aparte de los usuarios inmediatos del sistema.

Una vez que se han identificado y estructurado los puntos de vista, se debe intentar identificar los más importantes y empezar con ellos al descubrir los requerimientos del sistema.

Entrevistas

Las entrevistas formales o informales con los stakeholders del sistema son parte de la mayoría de los procesos de la ingeniería de requerimientos. En estas entrevistas, el equipo de la ingeniería de requerimientos hace preguntas a los stakeholders sobre el sistema que utilizan y sobre el sistema a desarrollar. Los requerimientos provienen de las respuestas a estas preguntas. Las entrevistas pueden ser de dos tipos:

1. Entrevistas cerradas donde los stakeholders responden a un conjunto predefinido de preguntas.

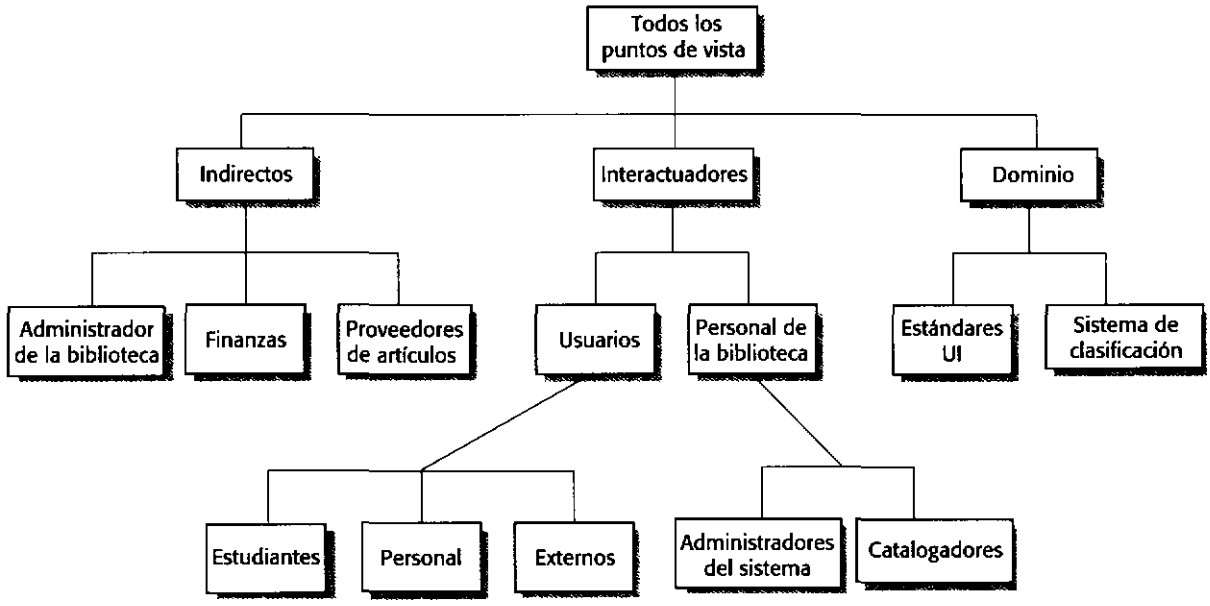


Figura 7.4 Puntos de vista en el LIBSYS.

2. Entrevistas abiertas donde no hay un programa predefinido. El equipo de la ingeniería de requerimientos examina una seria de cuestiones con los stakeholders del sistema y, por lo tanto, desarrolla una mejor comprensión de sus necesidades.

En la práctica, las entrevistas con los stakeholders normalmente son una mezcla de éstos. Las respuestas a algunas preguntas pueden conducir a otras cuestiones que se discuten de una forma menos estructurada. Las discusiones completamente abiertas raramente salen bien; la mayoría de las entrevistas requieren algunas preguntas para empezar y para mantener la entrevista centrada en el sistema a desarrollar.

Las entrevistas sirven para obtener una comprensión general de lo que hacen los stakeholders, cómo podrían interactuar con el sistema y las dificultades a las que se enfrentan con los sistemas actuales. A la gente le gusta hablar sobre su trabajo y normalmente se alegran de verse implicados en las entrevistas. Sin embargo, no son de tanta utilidad para la comprensión de los requerimientos del dominio de la aplicación.

Es difícil obtener conocimiento del dominio durante las entrevistas debido a dos razones:

1. Todos los especialistas de la aplicación utilizan terminología y jerga específica de un dominio. Es imposible para ellos discutir requerimientos del dominio sin utilizar esta terminología. Normalmente la utilizan de un modo preciso y sutil, por lo que es fácil que la malinterpreten los ingenieros de requerimientos.
2. Cierto conocimiento del dominio es tan familiar para los stakeholders que o lo encuentran difícil de explicar o piensan que es tan básico que no merece la pena mencionarlo. Por ejemplo, para un bibliotecario, es evidente que todas las adquisiciones se catalogan antes de agregarlas a la biblioteca. Sin embargo, esto puede no ser obvio para el entrevistador, por lo que no se tiene en cuenta en los requerimientos.

Las entrevistas no son una técnica eficaz para obtener conocimiento sobre los requerimientos y restricciones organizacionales debido a que existen sutiles poderes e influencias en-

tre los stakeholders en la organización. Las estructuras organizacionales publicadas rara vez se corresponden con la realidad de la toma de decisiones en una organización, pero los entrevistados pueden no desear revelar la estructura real en vez de la teórica a un desconocido. En general, la mayoría de la gente es reacia a discutir cuestiones políticas y organizacionales que pueden influir en los requerimientos.

Los entrevistadores eficaces tienen dos características:

1. No tienen prejuicios, evitan ideas preconcebidas sobre los requerimientos y están dispuestos a escuchar a los stakeholders. Si el stakeholder propone requerimientos sorprendentes, están dispuestos a cambiar su opinión del sistema.
2. Instan al entrevistado a empezar las discusiones con una pregunta, una propuesta de requerimientos o sugiriendo trabajar juntos en un prototipo del sistema. Decir a la gente «dime lo que quieres» es improbable que cause información de utilidad. La mayoría de la gente encuentra mucho más fácil hablar en un contexto definido en vez de en términos generales.

La información de la entrevistas complementa otras informaciones sobre el sistema de los documentos, observaciones de los usuarios, etcétera. Algunas veces, aparte de la información de los documentos, las entrevistas pueden ser la única fuente de información sobre los requerimientos del sistema. Sin embargo, las entrevistas por sí mismas tienden a omitir información esencial, por lo que deberían ser usadas al lado de otras técnicas de obtención de requerimientos.

Escenarios

Normalmente, las personas encuentran más fácil dar ejemplos de la vida real que descripciones abstractas. Pueden comprender y criticar un escenario de cómo podrían interactuar con un sistema software. Los ingenieros de requerimientos pueden utilizar la información obtenida de esta discusión para formular los requerimientos reales del sistema.

Los escenarios pueden ser especialmente útiles para agregar detalle a un esbozo de la descripción de requerimientos. Son descripciones de ejemplos de las sesiones de interacción. Cada escenario abarca una o más posibles interacciones. Se han desarrollado varias formas de escenarios, cada una de las cuales proporciona diferentes tipos de información en diferentes niveles de detalle sobre el sistema. Utilizar escenarios para describir requerimientos es una parte fundamental de los métodos ágiles, como la programación extrema, que se aborda en el Capítulo 17.

El escenario comienza con un esbozo de la interacción y, durante la obtención, se agregan detalles para crear una descripción completa de esta interacción. De forma general, un escenario puede incluir:

1. Una descripción de lo que esperan el sistema y los usuarios cuando el escenario comienza.
2. Una descripción del flujo normal de eventos en el escenario.
3. Una descripción de lo que puede ir mal y cómo manejarlo.
4. Información de otras actividades que se podrían llevar a cabo al mismo tiempo.
5. Una descripción del estado del sistema cuando el escenario termina.

Es posible llevar a cabo de manera informal la obtención de requerimientos basada en escenarios cuando los ingenieros de requerimientos trabajan con los stakeholders en la identificación de escenarios y en la captura de detalles de dichos escenarios. Los escenarios se pue-

Suposición inicial: El usuario ha entrado en el sistema LIBSYS y ha localizado la revista que contiene el artículo.

Normal: El usuario selecciona el artículo a copiar. El sistema insta al usuario a proporcionar la información de suscriptor de la revista o a indicar un método de pago del artículo. El pago se puede efectuar mediante tarjeta de crédito o citando un número de cuenta.

Se le solicita entonces al usuario que rellene un formulario de derechos de autor que mantiene los detalles de la transacción y se envía al sistema LIBSYS.

Se verifica el formulario de derechos de autor y, si se aprueba, se descarga la versión en PDF del artículo al área de trabajo del LIBSYS en la computadora del usuario y se informa al usuario que está disponible. Se le pide al usuario que seleccione una impresora y se imprime una copia del artículo. Si el artículo es de «sólo impresión» se elimina del sistema del usuario una vez que éste ha confirmado que se ha completado la impresión.

Qué puede salir mal: El usuario puede rellenar el formulario de derechos de autor incorrectamente. En este caso, se le debe volver a mostrar el formulario para su corrección. Si el formulario reenviado todavía es incorrecto, se rechaza la petición del artículo por parte del usuario.

El sistema puede rechazar el pago, en cuyo caso se rechaza la petición del artículo.

La descarga del artículo puede fallar, haciendo que el sistema lo reintente hasta que lo consiga o hasta que el usuario termine la sesión.

Es posible que no pueda imprimir el artículo. Si el artículo no es de «sólo impresión», se mantendrá en el espacio de trabajo del LIBSYS. De lo contrario, el artículo se elimina y se le cargan a la cuenta del usuario los costes del artículo.

Otras actividades: Descargas simultáneas de otros artículos.

Estado del sistema a la finalización: El usuario está dentro del sistema. El artículo descargado se ha borrado del espacio de trabajo del LIBSYS si es de sólo impresión.

Figura 7.5 Escenario para la descarga de artículos en el LIBSYS.

den redactar como texto, complementados por diagramas, fotografías de las pantallas, etcétera. De forma alternativa, se puede adoptar un enfoque más estructurado, como los escenarios de evento o los casos de uso.

Como ejemplo de escenario de texto sencillo, considere cómo un usuario del sistema de biblioteca LIBSYS puede utilizar el sistema. Se muestra este escenario en la Figura 7.5. El usuario desea imprimir una copia personal de un artículo de una revista médica. Esta revista hace copias de artículos disponibles gratuitamente para los suscriptores, pero las personas que no están suscritas tienen que pagar una cantidad por artículo. El usuario conoce el artículo, el título y la fecha de publicación.

Casos de uso

Los casos de uso son una técnica que se basa en escenarios para la obtención de requerimientos que se introdujeron por primera vez en el método Objetory (Jacobsen *et al.*, 1993). Actualmente se han convertido en una característica fundamental de la notación de UML, que se utiliza para describir modelos de sistemas orientados a objetos. En su forma más simple, un caso de uso identifica el tipo de interacción y los actores involucrados. Por ejemplo, la Figura 7.6 muestra un caso de uso de alto nivel de la función de impresión de artículos en el LIBSYS descrita en la Figura 7.5.

La Figura 7.6 ilustra la esencia de la notación para los casos de uso. Los actores en el proceso se representan como figuras delineadas, y cada clase de interacción se representa como una elipse con su nombre. El conjunto de casos de uso representa todas las posibles interac-

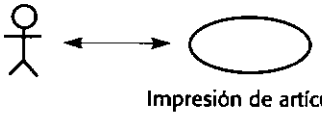


Figura 7.6 Un caso de uso sencillo para la impresión de artículos.

ciones a representar en los requerimientos del sistema. La Figura 7.7 extiende el ejemplo del LIBSYS y muestra otros casos de uso en ese entorno.

Algunas veces existe confusión sobre si un caso de uso es un escenario o, como sugiere Fowler (Fowler y Scott, 1997), un caso de uso encierra un conjunto de escenarios, y cada uno de éstos es un hilo único a través del caso de uso. Si un escenario incluye múltiples hilos, habrá un escenario para la interacción normal y escenarios adicionales para las posibles excepciones.

Los casos de uso identifican las interacciones particulares con el sistema. Pueden ser documentadas con texto o vinculadas a modelos UML que desarrollan el escenario en más detalle. Los diagramas de secuencia (introducidos en el Capítulo 6) se utilizan a menudo para añadir información a un caso de uso. Éstos muestran los actores involucrados en la interacción, los objetos con los que interactúan y las operaciones asociadas con estos objetos.

Como ejemplo de esto, la Figura 7.8 muestra las interacciones involucradas en la utilización del LIBSYS para la descarga e impresión de un artículo. En la Figura 7.8, existen cuatro objetos de clases —Artículo, Formulario, Espacio de trabajo e Impresora— involucradas en esta interacción. La secuencia de acciones es de arriba abajo, y las etiquetas de las fechas entre los actores y los objetos indican los nombres de las operaciones. Fundamentalmente, una petición de un artículo por parte de un usuario provoca una petición de un formulario de derechos de autor. Una vez que el usuario ha completado el formulario, se descarga el artículo y se envía a la impresora. Cuando termina la impresión, se elimina el artículo del espacio de trabajo del LIBSYS.

UML es un estándar *de facto* para el modelado orientado a objetos, por lo que los casos de uso y la obtención de requerimientos basada en casos de uso se utilizan cada vez más

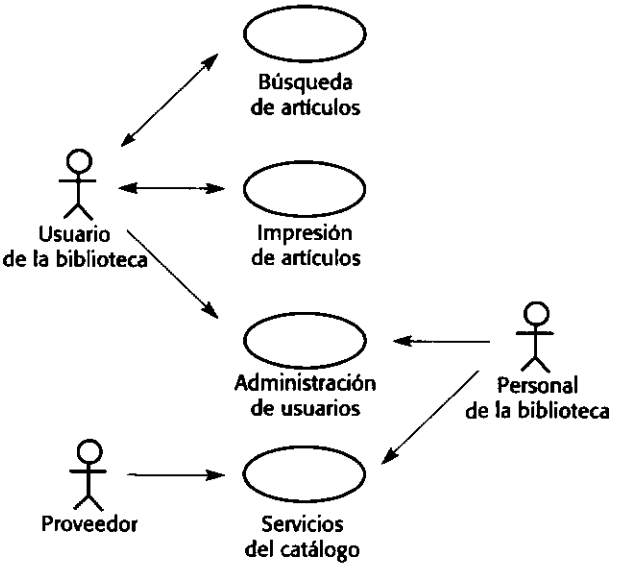


Figura 7.7 Casos de uso para el sistema de biblioteca.

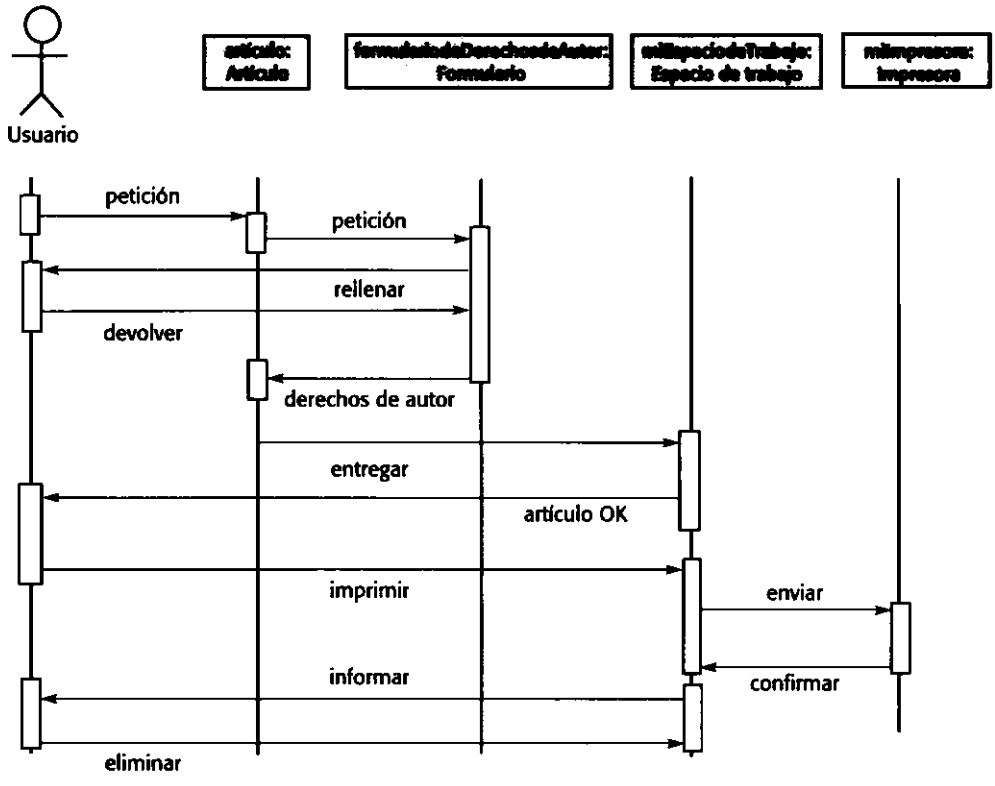


Figura 7.8 Interacciones del sistema para la impresión de artículos.

para la obtención de requerimientos. Se analizan otros tipos de modelos UML en el Capítulo 8, el cual trata el modelado de sistemas, y en el Capítulo 14, que aborda el diseño orientado a objetos.

Los escenarios y los casos de uso son técnicas eficaces para obtener requerimientos para los puntos de vista de los interactuadores, donde cada tipo de interacción se puede representar como un caso de uso. También se pueden utilizar conjuntamente con algunos puntos de vista indirectos cuando éstos reciben resultados (como un informe de gestión) del sistema. Sin embargo, debido a que se centran en las interacciones, no son tan eficaces para obtener restricciones y requerimientos de negocio y no funcionales de alto nivel de puntos de vista indirectos o para descubrir requerimientos del dominio.

7.2.2 Etnografía

Los sistemas software no existen de forma aislada: se utilizan en un contexto social y organizacional, y los requerimientos de sistemas software se pueden derivar y restringir según ese contexto. A menudo, satisfacer estos requerimientos sociales y organizacionales es crítico para el éxito del sistema. Una razón de por qué muchos sistemas software se entregan pero nunca se utilizan es que no se tiene en cuenta adecuadamente la importancia de este tipo de requerimientos del sistema.

La etnografía es una técnica de observación que se puede utilizar para entender los requerimientos sociales y organizacionales. Un analista se sumerge por sí solo en el entorno laboral donde se utilizará el sistema. Observa el trabajo diario y anota las tareas reales en las que los participantes están involucrados. El valor de la etnografía es que ayuda a los analistas a

descubrir los requerimientos implícitos que reflejan los procesos reales mas que los formales en los que la gente está involucrada.

A menudo, a la gente le resulta muy difícil articular detalles de su propio trabajo debido a que lo asumen como algo completamente natural. Comprenden su propio trabajo pero no su relación con las demás tareas en la organización. Los factores sociales y organizacionales que afectan al trabajo, pero que no son obvios para las personas, es posible que sólo queden claros cuando se fije en ellos un observador imparcial.

Suchman (Suchman, 1987) utilizó la etnografía para estudiar el trabajo de oficina y observó que las prácticas del trabajo real eran mucho más ricas, más complejas y más dinámicas que los modelos sencillos supuestos por los sistemas de automatización de oficinas. La diferencia entre el trabajo supuesto y el real fue la razón más importante de por qué estos sistemas de oficina no han tenido efectos significativos en la productividad. Otros estudios etnográficos para la comprensión de requerimientos del sistema se han llevado a cabo en sistemas de control del tráfico aéreo (Bentley *et al.*, 1992; Hughes *et al.*, 1993), salas de control del metro (Heath y Luff, 1992), sistemas financieros y varias actividades de diseño (Hughes *et al.*, 1994; Hughes *et al.*, 1994). En mi propia investigación, he examinado métodos de integración de la etnografía en el proceso de la ingeniería del software vinculándola a los métodos de la ingeniería de requerimientos (Viller y Sommerville, 1999; Viller y Sommerville, 1998; Viller y Sommerville, 2000) y documentando patrones de interacción en sistemas cooperativos (Martin *et al.*, 2001; Martin *et al.*, 2002; Martin y Sommerville, 2004).

La etnografía es especialmente efectiva para descubrir dos tipos de requerimientos:

1. *Los requerimientos que se derivan de la forma en la que la gente trabaja realmente* más que de la forma en la que las definiciones de los procesos establecen que debería trabajar. Por ejemplo, los controladores del tráfico aéreo pueden desconectar un sistema de alerta de conflictos que detecta aviones que cruzan las trayectorias de vuelo, aun cuando los procedimientos de control normales especifican que debe utilizarse. Su estrategia de control está diseñada para asegurar que estos aviones se separarán antes de que ocurran problemas y consideran que la alarma de alerta de conflictos los distrae de su trabajo.
2. *Los requerimientos que se derivan de la cooperación y conocimiento de las actividades de la gente.* Por ejemplo, los controladores del tráfico aéreo pueden utilizar el conocimiento del trabajo de otros controladores para predecir el número de aviones que entrará en su sector de control. Después modifican sus estrategias de control dependiendo del volumen de trabajo pronosticado. Por lo tanto, un sistema automático de control del tráfico aéreo debe permitir a los controladores en un sector tener alguna visibilidad del trabajo en los sectores adyacentes.

La etnografía se puede combinar con la construcción de prototipos (Figura 7.9). La etnografía suministra información al desarrollo del prototipo de forma que se requieren menos ci-

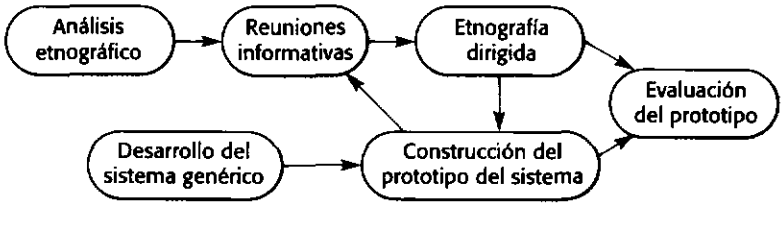


Figura 7.9 Etnografía y construcción de prototipos para el análisis de requerimientos.

clos de refinamiento. Además, la construcción de prototipos se centra en la etnografía para identificar problemas y preguntas que se puedan discutir posteriormente con el etnógrafo. Éste debe buscar las respuestas a estas preguntas durante la siguiente fase de estudio del sistema (Sommerville *et al.*, 1993).

Los estudios etnográficos pueden revelar los detalles de los procesos críticos que otras técnicas de obtención de requerimientos a menudo olvidan. Sin embargo, puesto que se centran en el usuario final, este enfoque no es apropiado para descubrir los requerimientos organizacionales o del dominio. Los estudios etnográficos no siempre pueden identificar nuevas propiedades que se deban agregar al sistema. Por lo tanto, la etnografía no es un enfoque completo para la obtención de requerimientos por sí mismo, y debe utilizarse para complementar otros enfoques, como el análisis de casos de uso.

7.3 Validación de requerimientos

La validación de requerimientos trata de mostrar que éstos realmente definen el sistema que el cliente desea. Coincide parcialmente con el análisis ya que éste implica encontrar problemas con los requerimientos. La validación de requerimientos es importante debido a que los errores en el documento de requerimientos pueden conducir a importantes costes al repetir el trabajo cuando son descubiertos durante el desarrollo o después de que el sistema esté en uso. El coste de arreglar un problema en los requerimientos haciendo un cambio en el sistema es mucho mayor que reparar los errores de diseño o los de codificación. La razón de esto es que un cambio en los requerimientos normalmente significa que el diseño y la implementación del sistema también deben cambiar y que éste debe probarse nuevamente.

Durante el proceso de validación de requerimientos, se deben llevar a cabo verificaciones sobre requerimientos en el documento de requerimientos. Estas verificaciones comprenden:

1. *Verificaciones de validez.* Un usuario puede pensar que se necesita un sistema para llevar a cabo ciertas funciones. Sin embargo, el razonamiento y el análisis pueden identificar que se requieren funciones adicionales o diferentes. Los sistemas tienen diversos stakeholders con diferentes necesidades, y cualquier conjunto de requerimientos es inevitablemente un compromiso en el entorno del stakeholder.
2. *Verificaciones de consistencia.* Los requerimientos en el documento no deben contradecirse. Esto es, no debe haber restricciones o descripciones contradictorias de la misma función del sistema.
3. *Verificaciones de completitud.* El documento de requerimientos debe incluir requerimientos que definan todas las funciones y restricciones propuestas por el usuario del sistema.
4. *Verificaciones de realismo.* Utilizando el conocimiento de la tecnología existente, los requerimientos deben verificarse para asegurar que se pueden implementar. Estas verificaciones también deben tener en cuenta el presupuesto y la confección de agendas para el desarrollo del sistema.
5. *Verificabilidad.* Para reducir la posibilidad de discusiones entre el cliente y el contratista, los requerimientos del sistema siempre deben redactarse de tal forma que sean verificables. Esto significa que debe poder escribir un conjunto de pruebas que demuestren que el sistema a entregar cumple cada uno de los requerimientos especificados.

Pueden utilizarse, en conjunto o de forma individual, varias técnicas de validación de requerimientos:

1. *Revisiones de requerimientos.* Los requerimientos son analizados sistemáticamente por un equipo de revisores. Este proceso se trata en la siguiente sección.
2. *Construcción de prototipos* En este enfoque de validación, se muestra un modelo ejecutable del sistema a los usuarios finales y a los clientes. Éstos pueden experimentar con este modelo para ver si cumple sus necesidades reales. En el Capítulo 17 se estudia la construcción de prototipos y sus técnicas.
3. *Generación de casos de prueba.* Los requerimientos deben poder probarse. Si las pruebas para éstos se conciben como parte del proceso de validación, a menudo revela los problemas en los requerimientos. Si una prueba es difícil o imposible de diseñar, normalmente significa que los requerimientos serán difíciles de implementar y deberían ser considerados nuevamente. Desarrollar pruebas para los requerimientos del usuario antes de que se escriba código es una parte fundamental de la programación extrema.

No deben subestimarse los problemas en la validación de requerimientos. Es difícil demostrar que un conjunto de requerimientos cumple las necesidades del usuario. Los usuarios deben imaginarse el sistema en funcionamiento y cómo éste encajaría en su trabajo. Para los informáticos expertos es difícil llevar a cabo este tipo de análisis abstracto, pero para los usuarios del sistema es aún más difícil. Como consecuencia, rara vez se encuentran todos los problemas en los requerimientos durante el proceso de validación de éstos. Es inevitable que haya cambios adicionales de requerimientos para corregir las omisiones y las malas interpretaciones después de que el documento de requerimientos haya sido aprobado.

7.3.1 Revisiones de requerimientos

Una revisión de requerimientos es un proceso manual que involucra a personas tanto de la organización del cliente como de la del contratista. Ellos verifican el documento de requerimientos en cuanto a anomalías y omisiones. El proceso de revisión se puede gestionar de la misma forma que las inspecciones de programas (véase el Capítulo 22). Alternativamente, se puede organizar como una actividad más amplia con diferentes personas que verifican diferentes partes del documento.

Las revisiones de requerimientos pueden ser informales o formales. Las informales sencillamente implican que los contratistas deben tratar los requerimientos con tantos stakeholders del sistema como sea posible. Es sorprendente la frecuencia con la que finaliza la comunicación entre los desarrolladores y los stakeholders del sistema después de la obtención de requerimientos y que no exista confirmación de que los requerimientos documentados son realmente lo que dijeron que deseaban los stakeholders.

Antes de llevar a cabo una reunión para una revisión formal, muchos problemas se pueden detectar simplemente hablando del sistema con los stakeholders.

En la revisión formal de requerimientos, el equipo de desarrollo debe «conducir» al cliente a través de los requerimientos del sistema, explicándole las implicaciones de cada requerimiento. El equipo de revisión debe verificar cada requerimiento para la consistencia además de verificar los requerimientos como un todo para la completitud. Los revisores también pueden comprobar la:

1. *Verificabilidad.* ¿Puede probarse el requerimiento de modo realista?

2. *Comprensibilidad.* ¿Las personas que adquieren el sistema o los usuarios finales comprenden correctamente el requerimiento?
3. *Rastreabilidad.* ¿Está claramente establecido el origen del requerimiento? Puede tener que volver a la fuente del requerimiento para evaluar el impacto del cambio. La rastreabilidad es importante ya que permite evaluar el impacto del cambio en el resto del sistema. Se trata esto con mayor detalle en la siguiente sección.
4. *Adaptabilidad.* ¿Es adaptable el requerimiento? Es decir, ¿puede cambiarse el requerimiento sin causar efectos de gran escala en los otros requerimientos del sistema?

Los conflictos, contradicciones, errores y omisiones en los requerimientos deben ser señalados por los revisores y registrarse formalmente en el informe de revisión. Queda en los usuarios, la persona que adquiere el sistema y el desarrollador de éste negociar una solución para estos problemas identificados.

7.4 Gestión de requerimientos

Los requerimientos para sistemas software grandes son siempre cambiantes. Una razón es que estos sistemas normalmente se desarrollan para abordar problemas «traviesos» (como se vio en el Capítulo 2). Debido a que el problema no puede definirse completamente, es muy probable que los requerimientos del software sean incompletos. Durante el proceso del software, la comprensión del problema por parte de los stakeholders está cambiando constantemente. Estos requerimientos deben entonces evolucionar para reflejar esta perspectiva cambiante del problema.

Además, una vez que un sistema se ha instalado, inevitablemente surgen nuevos requerimientos. Es difícil para los usuarios y clientes del sistema anticipar qué efectos tendrá el sistema nuevo en la organización. Cuando los usuarios finales tienen experiencia con un sistema, descubren nuevas necesidades y prioridades:

1. Normalmente, los sistemas grandes tienen una comunidad de usuarios diversa donde los usuarios tienen diferentes requerimientos y prioridades. Éstos pueden contradecirse o estar en conflicto. Los requerimientos finales del sistema son inevitablemente un compromiso entre ellos y, con la experiencia, a menudo se descubre que la ayuda suministrada a los diferentes usuarios tiene que cambiarse.
2. Las personas que pagan por el sistema y los usuarios de éste raramente son la misma persona. Los clientes del sistema imponen requerimientos debido a las restricciones organizacionales y de presupuesto. Éstos pueden estar en conflicto con los requerimientos de los usuarios finales y, después de la entrega, pueden tener que añadirse nuevas características de apoyo al usuario si el sistema tiene que cumplir sus objetivos.
3. El entorno de negocios y técnico del sistema cambia después de la instalación, y estos cambios se deben reflejar en el sistema. Se puede introducir nuevo hardware, puede ser necesario que el sistema interactúe con otros sistemas, las prioridades de negocio pueden cambiar con modificaciones consecuentes en la ayuda al sistema, y puede haber una nueva legislación y regulaciones que deben ser implementadas por el sistema.

La gestión de requerimientos es el proceso de comprender y controlar los cambios en los requerimientos del sistema. Es necesario mantenerse al tanto de los requerimientos particulares y mantener vínculos entre los requerimientos dependientes de forma que se pueda evaluar el impacto de los cambios en los requerimientos. Hay que establecer un proceso formal para

implementar las propuestas de cambios y vincular éstos a los requerimientos del sistema. El proceso de gestión de requerimientos debería empezar en cuanto esté disponible una versión preliminar del documento de requerimientos, pero se debería empezar a planificar cómo gestionar los requerimientos que cambian durante el proceso de obtención de requerimientos.

7.4.1 Requerimientos duraderos y volátiles

La evolución de los requerimientos durante el proceso de ingeniería de requerimientos y después de que un sistema esté en uso es inevitable. El desarrollo de requerimientos software centra su atención en las capacidades de éste, los objetivos del negocio y otros sistemas de la organización. Conforme se va desarrollando la definición de los requerimientos, normalmente tiene una mejor comprensión de las necesidades de los usuarios. Esto retroalimenta la información del usuario, quien puede entonces proponer un cambio en los requerimientos (Figura 7.10). Además, especificar y desarrollar un sistema grande puede llevar varios años. Durante ese tiempo, el entorno del sistema y los objetivos del negocio cambian, y los requerimientos evolucionan para reflejar esto.

Desde una perspectiva evolutiva, los requerimientos se dividen en dos clases:

1. *Requerimientos duraderos*. Son requerimientos relativamente estables que se derivan de la actividad principal de la organización y que están relacionados directamente con el dominio del sistema. Por ejemplo, en un hospital siempre habrá requerimientos que se refieren a los pacientes, médicos, enfermeras y tratamientos. Estos requerimientos se pueden derivar de los modelos del dominio que muestran las entidades y relaciones que caracterizan un dominio de aplicación (Easterbrook, 1993; Prieto-Díaz y Arango, 1991).
2. *Requerimientos volátiles*. Son requerimientos que probablemente cambian durante el proceso de desarrollo del sistema o después de que éste se haya puesto en funcionamiento. Un ejemplo serían los requerimientos resultantes de las políticas gubernamentales sobre sanidad.

Harker y otros (Harker *et al.*, 1993) han indicado que los requerimientos volátiles se dividen en cinco clases. Utilizando éstas como base, se ha desarrollado la clasificación mostrada en la Figura 7.11.

7.4.2 Planificación de la gestión de requerimientos

La planificación es una primera etapa esencial del proceso de gestión de requerimientos. La gestión de requerimientos tiene un coste elevado. Para cada proyecto, la etapa de planifica-

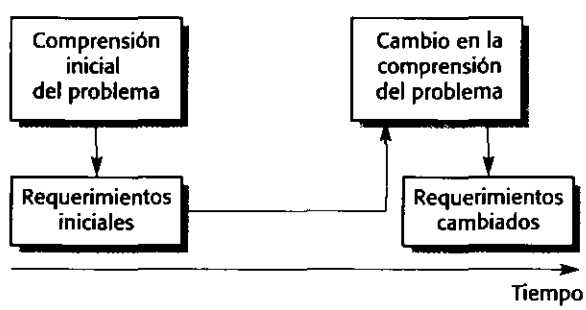


Figura 7.10 Evolución de los requerimientos.

Requerimientos cambiantes	**Requerimientos que cambian debido a los cambios en el entorno en el que opera la organización. Por ejemplo, en los sistemas hospitalarios, el pago del cuidado del paciente puede cambiar y así requerir un tratamiento diferente la información a recopilar.**
Requerimientos emergentes	**Requerimientos que emergen al incrementarse la comprensión del cliente en el desarrollo del sistema. El proceso de diseño puede revelar requerimientos emergentes nuevos.**
Requerimientos consecuentes	**Requerimientos que son resultado de la introducción del sistema informático. Esta introducción puede cambiar los procesos de la organización y desarrollar nuevas formas de trabajar que generan nuevos requerimientos del sistema.**
Requerimientos de compatibilidad	**Requerimientos que dependen de sistemas particulares o procesos de negocios dentro de la organización. Cuando estos últimos cambian, los requerimientos de compatibilidad del sistema encargado o entregado también pueden tener que cambiar.**

Figura 7.11 Clasificación de los requerimientos volátiles.

ción establece el nivel de detalle necesario en la gestión de requerimientos. Durante la etapa de gestión de requerimientos, habrá que decidir sobre:

1. *La identificación de requerimientos.* Cada requerimiento se debe identificar de forma única de tal forma que puedan ser remitidos por otros requerimientos de manera que pueda utilizarse en las evaluaciones de rastreo.
2. *Un proceso de gestión del cambio.* Éste es el conjunto de actividades que evalúan el impacto y coste de los cambios. Se trata con mayor detalle este proceso en la sección siguiente .
3. *Políticas de rastreo.* Estas políticas definen las relaciones entre los requerimientos, y entre éstos y el diseño del sistema que se debe registrar y la manera en que estos registros se deben mantener.
4. *Ayuda de herramientas CASE.* La gestión de requerimientos comprende el procesamiento de grandes cantidades de información sobre los requerimientos. Las herramientas que se pueden utilizar van desde sistemas de gestión de requerimientos especializados hasta hojas de cálculo y sistemas sencillos de bases de datos.

Existen muchas relaciones entre los requerimientos y entre éstos y el diseño del sistema. También existen vínculos entre los requerimientos y las razones fundamentales por las que éstos se propusieron. Cuando se proponen cambios, se debe rastrear el impacto de estos cambios en los otros requerimientos y en el diseño del sistema. El rastreo es una propiedad de la especificación de requerimientos que refleja la facilidad de encontrar requerimientos relacionados.

Existen tres tipos de información de rastreo que pueden ser mantenidos:

1. *La información de rastreo de la fuente* vincula los requerimientos con los stakeholders que propusieron los requerimientos y la razón de éstos. Cuando se propone un cambio, esta información se utiliza para encontrar y consultar a los stakeholders sobre el cambio.
2. *La información de rastreo de los requerimientos* vincula los requerimientos dependientes en el documento de requerimientos. Esta información se utiliza para evaluar cómo es probable que muchos requerimientos se vean afectados por un cambio propuesto y la magnitud de los cambios consecuentes en los requerimientos.

1.1			D	R				
1.2				D			R	D
1.3		R			R			
2.1				R		D		D
2.2								D
2.3			R		D			
3.1								R
3.2							R	

Figura 7.12 Una matriz de rastreo.

3. *La información de rastreo del diseño* vincula los requerimientos a los módulos del diseño en los cuales son implementados. Esta información se utiliza para evaluar el impacto de los cambios de los requerimientos propuestos en el diseño e implementación del sistema.

A menudo, la información de rastreo se representa utilizando matrices de rastreo, las cuales relacionan los requerimientos con los stakeholders, con los módulos del diseño o los requerimientos entre ellos. En una matriz de rastreo de requerimientos, cada requerimiento se representa en una fila y en una columna de la matriz. Cuando existen dependencias entre diferentes requerimientos, éstas se registran en la celda en la intersección fila/columna.

La Figura 7.12 muestra una matriz de rastreo sencilla que registra las dependencias entre los requerimientos. Una «D» en la intersección fila/columna ilustra que el requerimiento en la fila depende del requerimiento señalado en la columna; una «R» significa que existe alguna otra relación más débil entre los requerimientos. Por ejemplo, ambos pueden definir los requerimientos para partes del mismo subsistema.

Las matrices de rastreo pueden utilizarse cuando se tiene que gestionar un número pequeño de requerimientos, pero son difíciles de manejar y caras de mantener para sistemas grandes con muchos requerimientos. Para estos sistemas, se debería captar la información de rastreo en una base de datos de requerimientos en la que cada requerimiento esté explícitamente vinculado a los requerimientos relacionados. De esta forma, se puede evaluar el impacto de los cambios utilizando las facilidades de exploración de la base de datos. Se pueden generar automáticamente matrices de rastreo de la base de datos.

La gestión de requerimientos necesita ayuda automatizada; las herramientas CASE para esto deben elegirse durante la fase de planificación. Se precisan herramientas de ayuda para:

1. *Almacenar requerimientos.* Los requerimientos deben mantenerse en un almacén de datos seguro y administrado que sea accesible a todos los que estén implicados en el proceso de ingeniería de requerimientos.
2. *Gestionar el cambio.* Este proceso (Figura 7.13) se simplifica si está disponible una herramienta de ayuda.

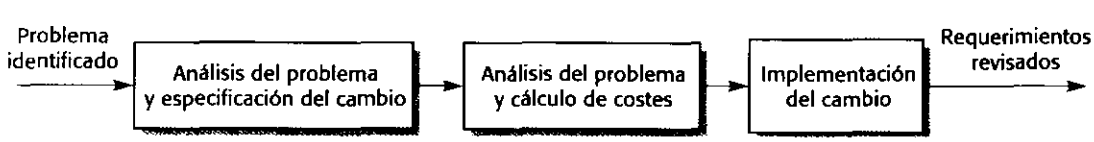


Figura 7.13 Gestión de cambios en los requerimientos.

3. *Gestionar el rastreo.* Como se indicó anteriormente, las herramientas de ayuda para el rastreo permiten que se descubran requerimientos relacionados. Algunas herramientas utilizan técnicas de procesamiento del lenguaje natural para ayudarle a descubrir posibles relaciones entre los requerimientos.

Para sistemas pequeños, es posible que no sea necesario utilizar herramientas de gestión de requerimientos especializadas. El proceso de gestión de requerimientos puede llevarse a cabo utilizando los recursos disponibles en los procesadores de texto, hojas de cálculo y bases de datos en PC. Sin embargo, para sistemas grandes, se requieren herramientas de ayuda más especializadas. En el sitio web del libro he incluido enlaces a herramientas de gestión de requerimientos como DOORS y RequisitePro.

7.4.3 Gestión del cambio de los requerimientos

La gestión del cambio de los requerimientos (Figura 7.13) se debe aplicar a todos los cambios propuestos en los requerimientos. La ventaja de utilizar un proceso formal para gestionar el cambio es que todos los cambios propuestos son tratados de forma consistente y que los cambios en el documento de requerimientos se hacen de forma controlada. Existen tres etapas principales en un proceso de gestión de cambio:

1. *Análisis del problema y especificación del cambio.* El proceso empieza con la identificación de un problema en los requerimientos o, algunas veces, con una propuesta de cambio específica. Durante esta etapa, el problema o la propuesta de cambio se analiza para verificar que ésta es válida. Los resultados del análisis se pasan al solicitante del cambio, y algunas veces se hace una propuesta de cambio de requerimientos más específica.
2. *Análisis del cambio y cálculo de costes.* El efecto de un cambio propuesto se valora utilizando la información de rastreo y el conocimiento general de los requerimientos del sistema. El coste de hacer un cambio se estima en términos de modificaciones al documento de requerimientos y, si es apropiado, al diseño e implementación del sistema. Una vez que este análisis se completa, se toma una decisión sobre si se continúa con el cambio de requerimientos.
3. *Implementación del cambio.* Se modifica el documento de requerimientos y, en su caso, el diseño e implementación del sistema. Debe organizar el documento de requerimientos de modo que pueda hacer cambios en él sin tener que hacer grandes reorganizaciones o redactar nuevamente gran cantidad del mismo. Como sucede con los programas, los cambios en los documentos se llevan a cabo minimizando las referencias externas y haciendo las secciones del documento tan modulares como sea posible. De esta manera, se pueden cambiar y reemplazar secciones individuales sin afectar a otras partes del documento.

Si se requiere de forma urgente un cambio en los requerimientos del sistema, existe siempre la tentación de hacer ese cambio al sistema y entonces modificar de forma retrospectiva el documento de requerimientos. Esto conduce casi inevitablemente a que la especificación de requerimientos y la implementación del sistema se desfasen. Una vez que se han hecho los cambio en el sistema, los del documento de requerimientos se pueden olvidar o se hacen de forma que no concuerdan con los cambios del sistema.

Los procesos de desarrollo iterativo, como la programación extrema, se han diseñado para hacer frente a los requerimientos que cambian durante el proceso de desarrollo. En estos pro-

cesos, cuando un usuario propone un cambio en los requerimientos, no se hace a través de un proceso formal de gestión del cambio. Más bien, el usuario tiene que establecer la prioridad del cambio y, si es de alta prioridad, decidir qué característica del sistema que fue planificada para la siguiente iteración debería abandonarse.

PUNTOS CLAVE

- **El proceso de ingeniería de requerimientos incluye un estudio de viabilidad, así como la obtención, análisis, especificación, validación y gestión de requerimientos.**
- **La obtención y análisis de requerimientos es un proceso iterativo que puede ser representado como una espiral de actividades —el descubrimiento de requerimientos, la clasificación y organización de requerimientos, la negociación de requerimientos y la documentación de requerimientos.**
- **Los diferentes stakeholders del sistema tienen requerimientos diferentes. Por lo tanto, todos los sistemas complejos deben analizarse desde varios puntos de vista. Éstos pueden ser personas u otros sistemas que interactúan con el sistema a especificar, stakeholders afectados por el sistema, o puntos de vista del dominio que restringen los requerimientos.**
- **Los factores sociales y organizacionales tienen una fuerte influencia sobre los requerimientos del sistema y pueden determinar si el software realmente se utiliza.**
- **La validación de requerimientos es el proceso de verificar los requerimientos en cuanto a validez, consistencia, completitud, realismo y verificabilidad. Las principales técnicas para la validación son las revisiones de los requerimientos y la construcción de prototipos.**
- **Los cambios en los negocios, organizacionales y técnicos inevitablemente conducen a cambios en los requerimientos de un sistema software. La gestión de requerimientos es el proceso de gestionar y controlar estos cambios.**
- **El proceso de gestión de requerimientos incluye la gestión de la planificación, en la cual se diseñan las políticas y procedimientos para la gestión de requerimientos, y la del cambio, en la que usted analiza los cambios propuestos en los requerimientos y evalúa su impacto.**

LECTURAS ADICIONALES

«Requirements engineering». Este número especial incluye dos artículos que se centran en la ingeniería de requerimientos para dominios particulares (coches y dispositivos médicos) que ofrecen perspectivas interesantes en los procesos de la ingeniería de requerimientos en estas áreas. [*IEEE Software*, 20 (1), enero-febrero de 2003.]

Mastering the Requirements Process. Un libro interesante dirigido a los ingenieros de requerimientos que ejercen. Da consejos específicos para desarrollar un proceso de ingeniería de requerimientos efectivo. (S. Robertson y J. Robertson, 1999, Addison-Wesley.)

Requirements Engineering: Proceses and Techniques. Este libro incluye una visión más detallada de las actividades del proceso de ingeniería de requerimientos y analiza el método VORD y su aplicación. (G. Kotonya e I. Sommerville, 1998, John Wiley & Sons.)

EJERCICIOS

7.1 Mencione quiénes podrían ser los stakeholders en un sistema de registro de estudiantes universitarios. Explique por qué es casi inevitable que los requerimientos de diferentes stakeholders entren en conflicto de alguna forma.

7.2 Un sistema software se desarrolla para gestionar los registros de los pacientes que ingresan en una clínica para tratamiento. Los registros incluyen anotaciones de todos los controles habituales a los pacientes (temperatura, tensión arterial, etc.), los tratamientos dados, las reacciones de los pacientes, etcétera. Después del tratamiento, los registros de su estancia se envían al doctor del paciente, quien mantiene su historial clínico completo. Identifique los puntos de vista principales que se pueden tener en cuenta en la especificación del sistema y organícelos utilizando un diagrama de jerarquía de puntos de vista.

7.3 Para tres de los puntos de vista identificados en el sistema de biblioteca, LIBSYS (Figura 7.4), mencione tres requerimientos que podrían ser sugeridos por los stakeholders relacionados con ese punto de vista.

7.4 El sistema LIBSYS tiene que incluir soporte para la catalogación de nuevos documentos donde el catálogo del sistema puede ser distribuido a través de varias máquinas. ¿Cuáles son probablemente los tipos más importantes de requerimientos no funcionales relacionados con los servicios de catalogación?

7.5 Utilizando su conocimiento de cómo funciona un cajero automático de un banco, desarrolle un conjunto de casos de uso que podrían servir como una base para entender los requerimientos de un sistema de un cajero automático.

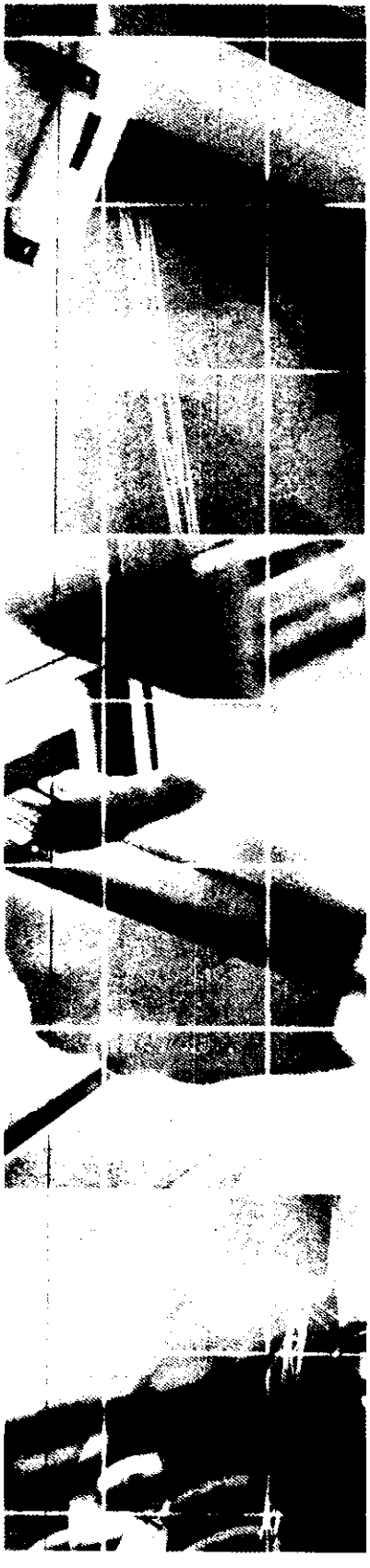
7.6 Dé un ejemplo de un tipo de sistema en el que los factores sociales y políticos pueden influir fuertemente en los requerimientos del sistema. Explique por qué estos factores son importantes en el ejemplo.

7.7 ¿Quiénes deberían estar implicados en la revisión de requerimientos? Establezca un modelo del proceso que muestre cómo se puede organizar una revisión de requerimientos.

7.8 ¿Por qué las matrices de rastreo son difíciles de manejar cuando existen muchos requerimientos en el sistema? Diseñe un mecanismo de estructuración de requerimientos, basado en puntos de vista, que pueda ayudar a reducir el tamaño del problema.

7.9 Cuando se hacen cambios de emergencia en los sistemas, el sistema software puede tener que modificarse antes de que los cambios en los requerimientos se aprueben. Sugiera un modelo de proceso para hacer estas modificaciones que asegure que el documento de requerimientos y la implementación del sistema no sean incompatibles.

7.10 Su compañía utiliza un método de análisis estándar que normalmente se aplica en todos los análisis de requerimientos. En su trabajo, comprueba que este método no puede representar factores sociales que son significativos en el sistema que usted analiza. Le señala esto a su jefe, quien le indica claramente que el estándar debe seguirse. Mencione qué debe hacer en tal situación.



8 Modelos del sistema

Objetivos

El objetivo de este capítulo es introducir varios modelos de sistemas que pueden desarrollarse durante el proceso de ingeniería de requerimientos. Cuando haya leído este capítulo:

- comprenderá por qué es importante establecer los límites de un sistema y modelar su contexto;
- comprenderá los conceptos del modelado del comportamiento, modelado de los datos y modelado de los objetos;
- habrá sido introducido en alguna de las notaciones definidas en el Lenguaje Unificado de Modelado (UML) y cómo estas notaciones pueden usarse para desarrollar modelos de sistemas.

Contenidos

Los requerimientos del usuario deberían redactarse en lenguaje natural debido a que tienen que ser comprendidos por personas que no son técnicos expertos. Sin embargo, pueden expresarse requerimientos del sistema más detallados de forma más técnica. Una técnica ampliamente usada es documentar la especificación del sistema como un conjunto de modelos del sistema. Estos modelos son representaciones gráficas que describen los procesos del negocio, el problema a resolver y el sistema que tiene que ser desarrollado. Debido a las representaciones gráficas usadas, los modelos son a menudo más comprensibles que las descripciones detalladas en lenguaje natural de los requerimientos del sistema. Ellos constituyen también un puente importante entre el proceso de análisis y diseño.

Pueden usarse modelos en el proceso de análisis para comprender el sistema existente que debe ser reemplazado o mejorado, o para especificar el nuevo sistema que sea requerido. Pueden desarrollarse diferentes modelos para representar el sistema desde diferentes perspectivas. Por ejemplo:

1. Una perspectiva externa, en la que se modela el contexto o entorno del sistema.
2. Una perspectiva de comportamiento, en la que se modela el comportamiento del sistema.
3. Una perspectiva estructural, en la que se modela la arquitectura del sistema o la estructura de los datos procesados por el sistema.

Estas tres perspectivas se tratan en este capítulo y también se estudia el modelado de objetos, que combina, de alguna manera, el modelado del comportamiento y de la estructura.

El aspecto más importante de un modelo del sistema es que omite los detalles. Un modelo del sistema es una abstracción del sistema que se está estudiando en lugar de una representación alternativa de ese sistema. Idealmente, una *representación* de un sistema debería mantener toda la información sobre la entidad que se está representando. Una *abstracción* simplifica y resalta de forma deliberada las características más relevantes. Por ejemplo, en el caso improbable de que este libro fuese publicado por capítulos en un periódico, la presentación en este caso sería una abstracción de los puntos clave del libro. Si fuese traducido del inglés al italiano, ésta podría ser una representación alternativa. La intención del traductor debería ser mantener toda la información tal y como se presenta en inglés.

Diferentes tipos de modelos del sistema se basan en distintas aproximaciones de abstracción. Un modelo de flujo de datos (por ejemplo) se centra en el flujo de datos y las transformaciones funcionales sobre esos datos. Se omiten los detalles de las estructuras de datos. Por el contrario, un modelo de entidades de datos y sus relaciones documentan las estructuras de datos del sistema en lugar de su funcionalidad.

Ejemplos de tipos de modelos del sistema que podrían crearse durante el proceso de análisis son:

1. *Un modelo de flujo de datos.* Los modelos de flujo de datos muestran cómo se procesan los datos en el sistema en diferentes etapas.
2. *Un modelo de composición.* Un modelo de composición o *agregación* muestra cómo las entidades del sistema están compuestas por otras entidades.
3. *Un modelo arquitectónico.* Los modelos arquitectónicos muestran los principales subsistemas que componen un sistema.
4. *Un modelo de clasificación.* Los diagramas de clases/herencia de objetos muestran cómo las entidades tienen características comunes.
5. *Un modelo de estímulo-respuesta.* Un modelo de estímulo respuesta o *diagrama de transición de estados* muestra cómo reacciona el sistema a eventos internos y externos.

Todos estos tipos de modelos se tratan en este capítulo. Siempre que es posible, se usan notaciones del Lenguaje Unificado de Modelado (UML), que se ha convertido en un lenguaje de modelado estándar para el modelado orientado a objetos (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a). Se usan notaciones intuitivas simples para la descripción del modelo en donde UML no incluye las notaciones adecuadas. Se está desarrollando una nueva versión de UML (UML 2.0), pero no estaba disponible en el momento de escribir este capítulo. Sin embargo, la notación UML utilizada aquí es compatible probablemente con UML 2.0.

8.1 Modelos de contexto

En una de las primeras etapas de la obtención de requerimientos y del proceso de análisis se deben definir los límites del sistema. Esto comprende trabajar conjuntamente con los *stakeholders* del sistema para distinguir lo que es el sistema y lo que es el entorno del sistema. Usted debería tomar estas decisiones al inicio del proceso para delimitar los costes del sistema y el tiempo necesario para el análisis.

En algunos casos, el límite entre un sistema y su entorno está relativamente claro. Por ejemplo, en el caso de que un sistema automático reemplace a un sistema existente manual o computerizado, el entorno del nuevo sistema es normalmente el mismo que el entorno del sistema existente. En otros casos, hay más flexibilidad, y usted decide lo que constituye el límite entre el sistema y su entorno durante el proceso de ingeniería de requerimientos.

Por ejemplo, suponga que está desarrollando la especificación para el sistema de biblioteca LIBSYS. Recuerde que este sistema pretende entregar versiones electrónicas de material con *derechos de autor* a los usuarios de computadoras. A continuación los usuarios pueden imprimir copias personales del material. Cuando desarrolla la especificación para este sistema, usted tiene que decidir si otros sistemas de bases de datos de bibliotecas tales como catálogos de bibliotecas están dentro de los límites del sistema. Si lo están, entonces puede permitir el acceso al sistema a través de la interfaz de usuario del catálogo; si no lo están, entonces los usuarios sufrirán la incomodidad de tenerse que mover de un sistema a otro.

La definición de un límite del sistema no es una decisión arbitraria. Aspectos sociales y organizacionales pueden implicar que la situación de los límites de un sistema puedan ser determinados por factores no técnicos. Por ejemplo, un límite de un sistema puede situarse de forma que el proceso de análisis sólo se pueda llevar a cabo en un lugar; puede ser elegido para que un gestor problemático no necesite ser consultado; puede situarse para que el coste del sistema se incremente, y la división del desarrollo del sistema debe, por lo tanto, expandirse al diseño e implementación del sistema.

Una vez que se han tomado algunas decisiones sobre los límites del sistema, parte de la actividad del análisis es la definición de ese contexto y de las dependencias que el sistema tiene sobre su entorno. Normalmente, la producción de un modelo arquitectónico sencillo es el primer paso en esta actividad.

La Figura 8.1 es un modelo arquitectónico que ilustra la estructura del sistema de información que incluye una red de cajeros automáticos. Los modelos arquitectónicos de alto nivel se expresan normalmente como sencillos diagramas de bloques en donde cada subsistema se representa por un rectángulo con nombre, y las líneas indican asociaciones entre los subsistemas.

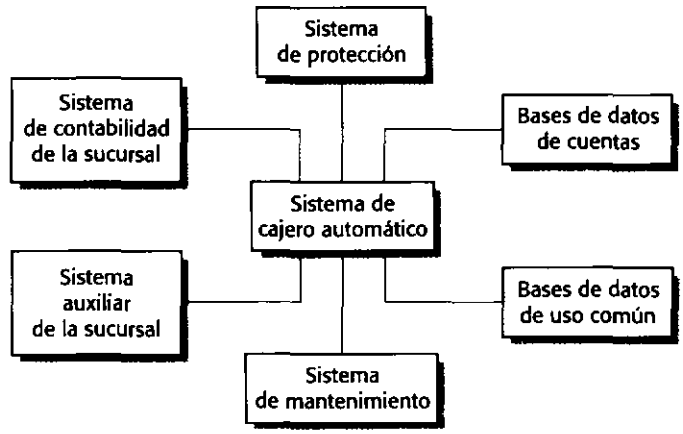


Figura 8.1
El contexto de un sistema ATM.

En la Figura 8.1, podemos ver que cada ATM (cajero automático) está conectado a una base de datos de cuentas, a un sistema de contabilidad de la sucursal, a un sistema de seguridad y a otro de mantenimiento de los cajeros. El sistema también está conectado a una base de datos de uso común que monitoriza cómo se usa la red de ATMs y también está conectado a un sistema auxiliar local de una sucursal. Este sistema auxiliar proporciona servicios tales como copias de seguridad e impresión. Éstos, por lo tanto, no necesitan incluirse en el propio sistema ATM.

Los modelos arquitectónicos describen el entorno de un sistema. Sin embargo, no muestran las relaciones entre otros sistemas del entorno y el sistema que se está especificando. Los sistemas externos podrían producir o consumir datos del sistema. Podrían compartir datos con el sistema, o podrían ser conectados directamente, conectados a través de una red o no estar conectados. Podrían estar físicamente en el mismo lugar o localizados en edificios diferentes. Todas estas relaciones podrían afectar a los requerimientos del sistema que se está definiendo y deben tenerse en cuenta.

Por lo tanto, los modelos arquitectónicos sencillos normalmente se complementan con otros modelos, tales como modelos de procesos, que muestran las actividades de los procesos soportadas por el sistema. Los modelos de flujo de datos (descritos en la sección siguiente) también pueden usarse para mostrar los datos que son transferidos entre el sistema y otros sistemas de su entorno.

La Figura 8.2 ilustra un modelo de proceso de adquisición de equipos de una organización. Esto implica especificar el equipo requerido, buscar y elegir a los proveedores, pedir el equipo, recibir los equipos a su llegada y a continuación verificarlos. Cuando especifique el soporte informático para este proceso, usted tiene que decidir cuáles de esas actividades serán realmente informatizadas. El resto de las actividades están fuera de los límites del sistema. En la Figura 8.2, la línea discontinua encierra las actividades que están dentro de los límites del sistema.

8.2 Modelos de comportamiento

Los modelos de comportamiento se utilizan para describir el comportamiento del sistema en su totalidad. Aquí se analizan dos tipos de modelos de comportamiento: modelos de flujo de datos, que modelan el procesamiento de los datos en el sistema, y modelos de máquinas de

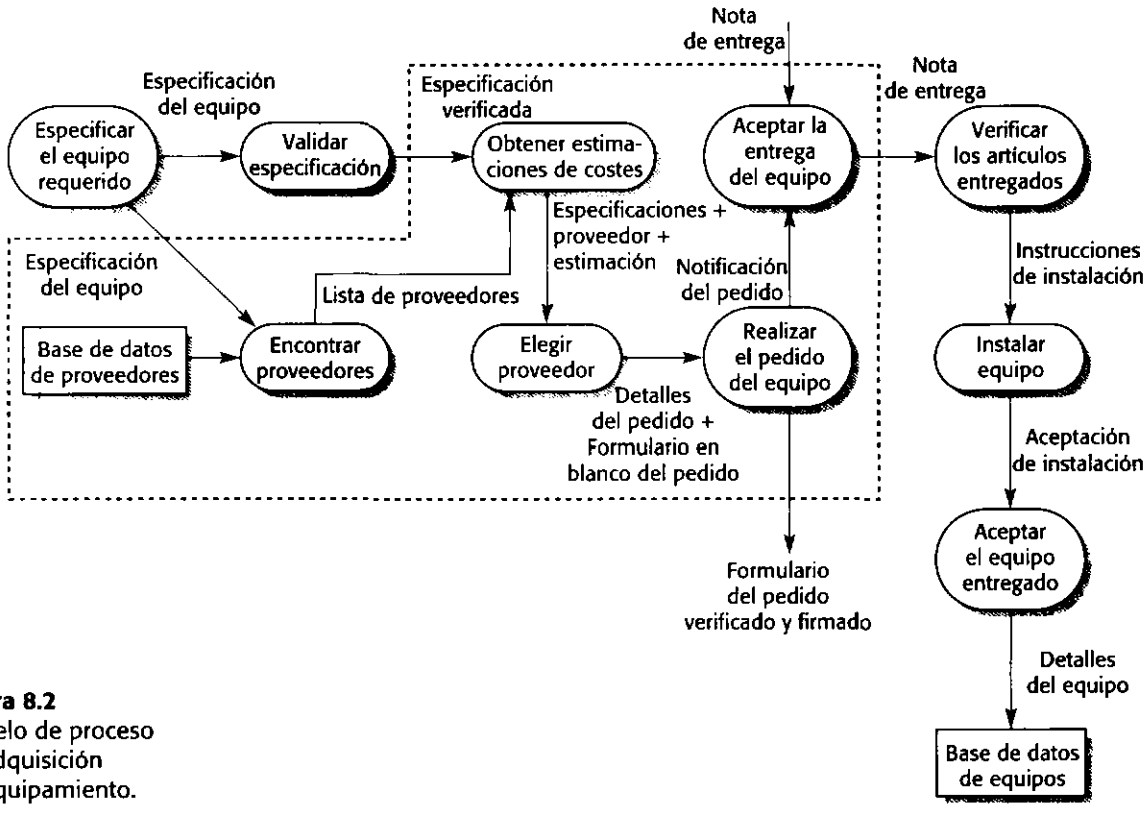


Figura 8.2
Modelo de proceso de adquisición de equipamiento.

estado, que modelan cómo el sistema reacciona a los eventos. Estos modelos pueden usarse de forma separada o conjuntamente, dependiendo del tipo de sistema que se esté desarrollando.

La mayoría de los sistemas de negocio están fundamentalmente dirigidos por los datos. Están controlados por las entradas de datos al sistema con relativamente poco procesamiento de eventos externos. Un modelo de flujo de datos puede ser todo lo que se necesite para representar el comportamiento de estos sistemas. Por el contrario, los sistemas de tiempo real a menudo están dirigidos por eventos con un mínimo procesamiento de datos. Un modelo de máquina de estados (analizado en la Sección 8.2.2) es la forma más efectiva de representar su comportamiento. Otras clases de sistemas pueden estar dirigidas tanto por datos como por eventos. En estos casos usted puede desarrollar ambos tipos de modelos.

8.2.1 Modelos de flujo de datos

Los modelos de flujo de datos son una forma intuitiva de mostrar cómo los datos son procesados por un sistema. A nivel de análisis, deberían usarse para modelar la forma en la que los datos son procesados en el sistema existente. El uso de modelos de flujo de datos para análisis comenzó a usarse ampliamente después de la publicación del libro de DeMarco (DeMarco, 1978) sobre análisis de sistemas estructurados. Estos modelos son una parte intrínseca de los métodos estructurados que han sido desarrollados a partir de este trabajo. La notación usada en ellos representa el procesamiento funcional (rectángulos redondeados), los almacenes de datos (rectángulos) y el flujo de datos entre funciones (flechas etiquetadas).

Los modelos de flujo de datos se utilizan para mostrar cómo fluyen los datos a través de una secuencia de pasos de procesamiento. Por ejemplo, un paso de procesamiento podría ser filtrar registros duplicados en una base de datos de clientes. Los datos se transforman en cada paso antes de moverse a la siguiente etapa. Estos pasos de procesamiento o transformaciones representan procesos software o funciones cuando los diagramas de flujo de datos se utilizan para documentar un diseño software. Sin embargo, en un modelo de análisis, el procesamiento se puede llevar a cabo por las personas o por las computadoras.

Un modelo de flujo de datos, que muestra los pasos que comprende el procesamiento de un pedido de productos (tales como equipamiento informático) en una organización, se ilustra en la Figura 8.3. Este modelo particular describe el procesamiento de datos en la actividad **Colocar el pedido del equipamiento** en el modelo completo del proceso mostrado en la Figura 8.2. El modelo muestra cómo el pedido para los productos fluye desde un proceso a otro. También muestra los almacenes de datos (**Fichero de pedidos** y **Fichero de presupuesto**) que están implicados en este proceso.

Los modelos de flujo de datos son valiosos debido a que realizan un seguimiento y documentan cómo los datos asociados con un proceso particular fluyen a través del sistema, y esto ayuda a los analistas a comprender el proceso. Los diagramas de flujo de datos tienen la ventaja de que, a diferencia de otras notaciones de modelado, son sencillos e intuitivos. Normalmente es posible explicarlos a los usuarios potenciales del sistema, quienes pueden entonces participar en la validación del análisis.

En principio, el desarrollo de modelos tales como modelos de flujo de datos debería ser un proceso «descendente». En este ejemplo, esto podría implicar que debería comenzarse analizando el proceso de adquisición de equipamiento en su totalidad. A continuación se seguiría con el análisis de subprocesos tales como el de solicitud de pedidos. En la práctica, el análisis nunca se hace así. Se analizan varios niveles al mismo tiempo. Los modelos de bajo nivel pueden desarrollarse primero y después abstraerse para crear un modelo más general.

Los modelos de flujo de datos muestran una perspectiva funcional en donde cada transformación representa un único proceso o función. Son particularmente útiles durante el análisis de requerimientos ya que pueden usarse para mostrar el procesamiento desde el principio hasta el final en un sistema. Es decir, muestra la secuencia completa de acciones que tienen lugar a partir de una entrada que se está procesando hasta la correspondiente salida que cons-

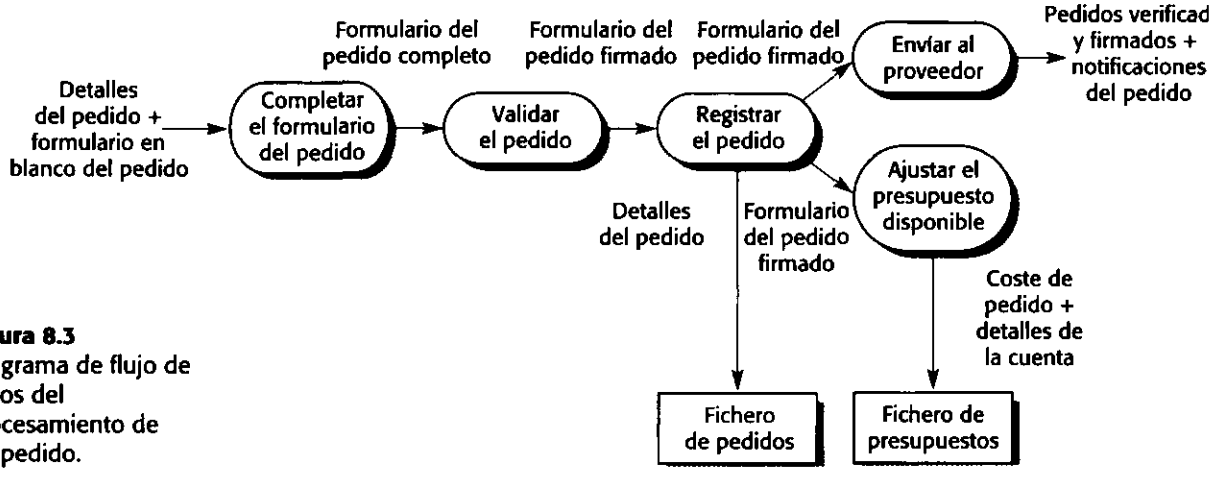


Figura 8.3
Diagrama de flujo de datos del procesamiento de un pedido.

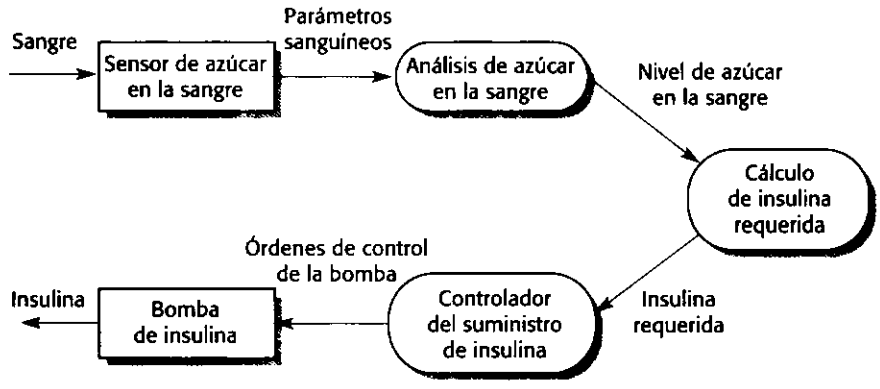


Figura 8.4 Diagrama de flujo de datos de una bomba de insulina.

tituye la respuesta del sistema. La Figura 8.4 ilustra este uso de los diagramas de flujo de datos. Éste es un diagrama del procesamiento que tiene lugar en el sistema de bomba de insulina introducido en el Capítulo 3.

8.2.2 Modelos de máquina de estados

Un modelo de máquina de estados describe cómo responde un sistema a eventos internos o externos. El modelo de máquina de estados muestra los estados del sistema y los eventos que provocan las transiciones de un estado a otro. No muestra el flujo de datos dentro del sistema. Este tipo de modelo se utiliza a menudo para modelar sistemas de tiempo real debido a que estos sistemas suelen estar dirigidos por estímulos procedentes del entorno del sistema. Por ejemplo, el sistema de alarma de tiempo real explicado en el Capítulo 13 responde a estímulos de sensores de movimiento, sensores de apertura de puertas, etcétera.

Los modelos de máquina de estados son una parte integral de los métodos de diseño de tiempo real tales como los propuestos por Ward y Mellor (Ward and Mellor, 1985), y Harel (Harel, 1987; Harel, 1988). El método de Harel usa una notación denominada *diagramas de estado* que fue la base para la notación del modelado de máquina de estados en UML.

Un modelo de máquina de estados de un sistema supone que, en cualquier momento, el sistema está en uno de varios estados posibles. Cuando se recibe un estímulo, éste puede disparar una transición a un estado diferente. Por ejemplo, un sistema de control de una válvula puede moverse desde un estado «Válvula abierta» a un estado «Válvula cerrada» cuando se reciba una orden (el estímulo) del operador.

Esta aproximación para el modelado de sistemas se ilustra en la Figura 8.5. Este diagrama muestra un modelo de máquina de estados de un sencillo horno microondas equipado con botones para fijar la potencia y el temporizador y para iniciar el sistema. Los hornos microondas reales son actualmente mucho más complejos que el sistema descrito aquí. Sin embargo, este modelo incluye las características esenciales del sistema. Para simplificar el modelo, se supone que la secuencia de acciones al usar el microondas es:

1. Seleccionar el nivel de potencia (ya sea media o máxima).
2. Introducir el tiempo de cocción.
3. Pulsar el botón de inicio, y la comida se cocina durante el tiempo establecido.

Por razones de seguridad, el horno no debería funcionar cuando la puerta esté abierta y, cuando se completa la cocción, suena un timbre. El horno dispone de una pantalla alfanumérica sencilla que se utiliza para visualizar varios mensajes de alerta y de precaución.

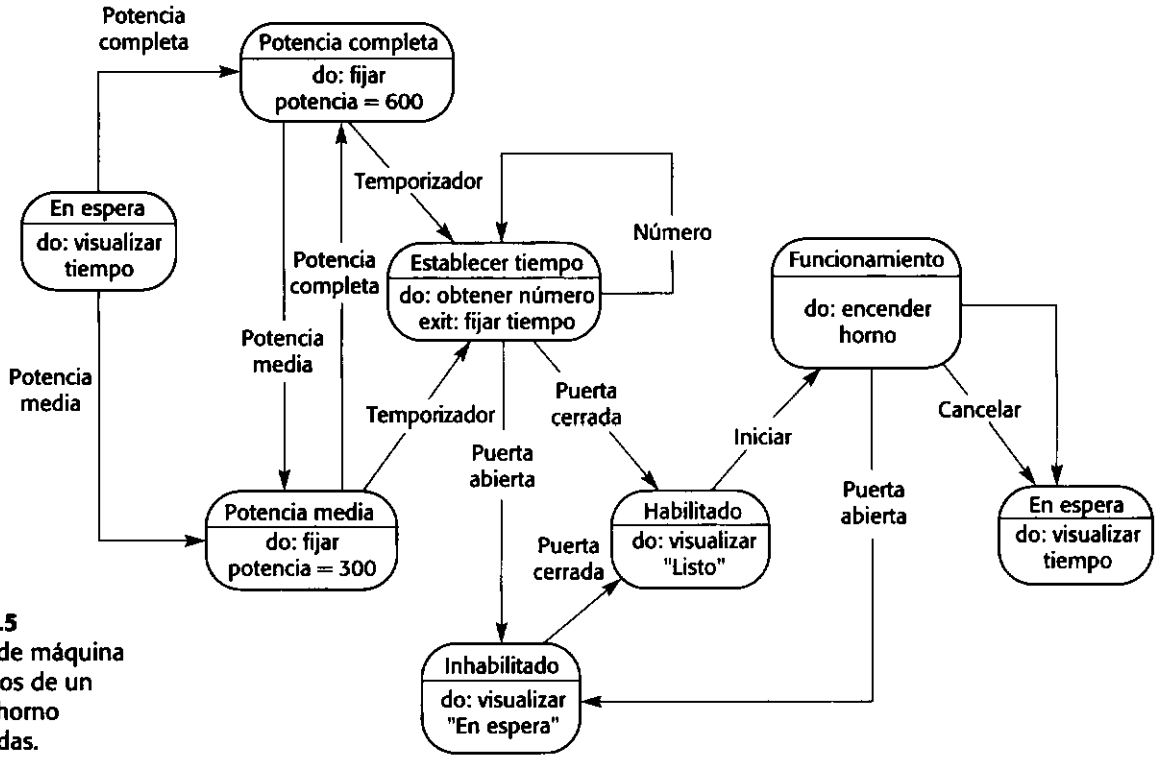


Figura 8.5
Modelo de máquina de estados de un sencillo horno microondas.

La notación UML utilizada para describir los modelos de máquina de estados está diseñada para modelar el comportamiento de los objetos. Sin embargo, es una notación de propósito general que se puede utilizar para cualquier tipo de modelado de máquina de estados. Los rectángulos redondeados en un modelo representan los estados del sistema. Incluyen una breve descripción (a continuación de «do») de las acciones realizadas en ese estado. Las flechas etiquetadas representan estímulos que fuerzan transiciones de un estado a otro.

Por lo tanto, en la Figura 8.5, podemos ver que el sistema responde bien inicialmente al botón de potencia máxima o al de potencia media. Los usuarios pueden cambiar de idea después de seleccionar uno de éstos y presionar el otro botón. Se fija el tiempo y, si la puerta está cerrada, se habilita el botón de comienzo. Pulsando este botón comienza a funcionar el horno y la cocción tiene lugar durante el tiempo especificado.

La notación UML indica la actividad que tiene lugar en un estado. En una especificación detallada del sistema, usted tiene que proporcionar más detalle sobre el estímulo y los estados del sistema (Figura 8.6). Esta información puede mantenerse en un diccionario de datos o enciclopedia (incluida en la Sección 8.3) y que es mantenida por las herramientas CASE utilizadas para crear el modelo del sistema.

El problema con la aproximación de la máquina de estados es que el número de posibles estados crece rápidamente. Por lo tanto, para los modelos de sistemas grandes, es necesaria una cierta estructuración de estos modelos de estados. Una forma de hacer esto es mediante la noción de un superestado que encierra a varios estados separados. Este superestado se asemeja a un único estado de un modelo de alto nivel, pero se expande a continuación con más detalle en un diagrama distinto. Para ilustrar este concepto, considere el estado **Funcionamiento** en la Figura 8.5. Éste es un superestado que puede expandirse, tal y como se ilustra en la Figura 8.7.

En espera	**El horno está esperando la entrada. La pantalla muestra la hora actual.**
Potencia media	**La potencia del horno se fija en 300 vatios. La pantalla muestra «Potencia media».**
Potencia máxima	**La potencia del horno se fija en 600 vatios. La pantalla muestra «Potencia máxima».**
Fijar tiempo	**Se fija el tiempo de cocción con el valor de entrada del usuario. La pantalla muestra el tiempo de cocción seleccionado y se actualiza en cuanto se fija el tiempo.**
Inhabilitado	**Se inhabilita el funcionamiento del horno por razones de seguridad. La luz interior del horno se enciende. La pantalla muestra «No listo».**
Habilitado	**Se habilita el funcionamiento del horno. Se apaga la luz de su interior. La pantalla muestra «Listo para cocinar».**
Funcionamiento	**El horno está en funcionamiento. La luz interior del horno se enciende. La pantalla muestra la cuenta atrás del temporizador. Cuando finaliza la cocción, suena un timbre durante 5 segundos. La luz interior se enciende. La pantalla muestra «Cocción finalizada» mientras el timbre suena.**
Estímulo	**Descripción**
Potencia media	**El usuario ha presionado el botón de potencia media.**
Potencia máxima	**El usuario ha presionado el botón de potencia máxima.**
Temporizador	**El usuario ha presionado uno de los botones del temporizador.**
Número	**El usuario ha presionado una tecla numérica.**
Puerta abierta	**El interruptor de la puerta del horno no está cerrado.**
Puerta cerrada	**El interruptor de la puerta del horno está cerrado.**
Iniciar	**El usuario ha presionado el botón de inicio.**
Cancelar	**El usuario ha presionado el botón de cancelar.**

Figura 8.6 Descripción de estados y estímulos para el horno microondas.

El estado Funcionamiento incluye varios subestados. Muestra que una operación comienza con una comprobación de estado, y que si se descubre cualquier problema, se activa una alarma y la operación queda inhabilitada. La cocción implica poner en marcha el generador de microondas durante el tiempo especificado; a su terminación, suena un timbre. Si la puerta se abre durante el funcionamiento, el sistema se mueve al estado inhabilitado, tal y como se muestra en la Figura 8.5.

8.3 Modelos de datos

La mayoría de los sistemas software grandes utilizan bases de datos de información de gran tamaño. En algunos casos, esta base de datos es independiente del sistema software. En otros, se crea para el sistema que se está desarrollando. Una parte importante del modelado de sistemas es la definición de la forma lógica de los datos procesados por el sistema. Éstos se denominan a menudo *modelos semánticos de datos*.

La técnica de modelado de datos más ampliamente usada es el modelado Entidad-Relación-Atributo (modelado ERA), que muestra las entidades de datos, sus atributos asociados y

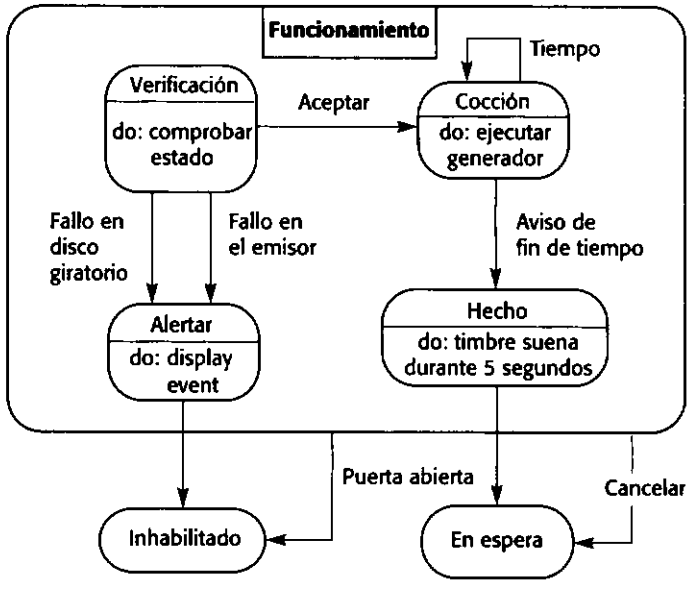


Figura 8.7 Funcionamiento del horno microondas.

las relaciones entre estas entidades. Esta aproximación de modelado fue propuesta por primera vez a mediados de los años 70 por Chen (Chen, 1976); desde entonces se han desarrollado varias variantes (Codd, 1979; Hammer y McLeod, 1981; Hull y King, 1987), y todas ellas tienen la misma forma básica.

Los modelos entidad-relación han sido ampliamente usados en el diseño de bases de datos. Los esquemas de bases de datos relacionales derivados de estos modelos se encuentran de manera natural en tercera forma normal, lo cual es una característica deseable (Barker, 1989). Debido al tipado explícito y al reconocimiento de subtipos y supertipos, también es sencillo implementar estos modelos utilizando bases de datos orientadas a objetos.

UML no incluye una notación específica para este modelado de bases de datos, ya que asume un proceso de desarrollo orientado a objetos y modela los datos utilizando objetos y sus relaciones. Sin embargo, se puede usar UML para representar un modelo semántico de datos. Se puede pensar en las entidades de un modelo ERA como clases de objetos simplificadas (no tienen operaciones), los atributos como atributos de la clase y las denominadas asociaciones entre clases como relaciones.

La Figura 8.8 es un ejemplo de un modelo de datos que es parte del sistema de biblioteca LIBSYS introducido en capítulos anteriores. Recuerde que LIBSYS se diseña para entregar copias de artículos con derechos de autor que han sido publicados en revistas y periódicos y para registrar el pago de estos artículos. Por lo tanto, el modelo de datos debe incluir información sobre el artículo, el poseedor de los derechos de autor y el comprador del artículo. Aquí hemos supuesto que estos pagos para los artículos no se hacen directamente sino a través de agencias nacionales de derechos de autor.

La Figura 8.8 muestra que un **Artículo** tiene atributos que representan el título, los autores, el nombre del fichero PDF del artículo y el honorario a pagar. Éste se enlaza con **Fuente**, en donde fue publicado el artículo, y con la **Agencia de derechos de autor** del país de publicación. Ambos, **Agencia de derechos de autor** y **Fuente**, se enlazan con **País**. El país de publicación es importante debido a que las leyes de derechos de autor varían de un país a otro. El diagrama también muestra que los **Compradores** emiten **Órdenes de compra** de **Artículos**.

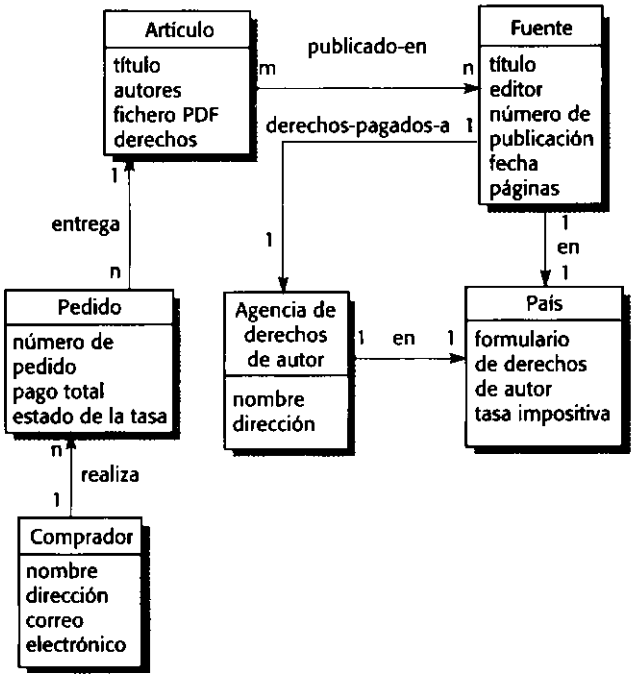


Figura 8.8
Modelo semántico de datos para el sistema LIBSYS.

Al igual que todos los modelos gráficos, a los modelos de datos les faltan detalles, y usted debería mantener descripciones más detalladas de las entidades, relaciones y atributos incluidas en el modelo. Usted puede reunir estas descripciones más detalladas en un repositorio o diccionario de datos. Los diccionarios de datos generalmente son útiles cuando desarrollamos modelos de sistemas y pueden utilizarse para gestionar toda la información de todos los tipos de modelos de sistemas.

Un diccionario de datos es, de forma simple, una lista de nombres ordenada alfabéticamente incluida en los modelos del sistema. El diccionario debería incluir, además del nombre, una descripción asociada de dicha entidad con nombre y, si el nombre representa un objeto compuesto, una descripción de la composición. Se puede incluir otra información, como la fecha de creación, el creador y la representación de la entidad dependiendo del tipo de modelo que se esté desarrollando.

Las ventajas de usar un diccionario de datos son las siguientes:

1. *Es un mecanismo para la gestión de nombres.* Muchas personas pueden tener que inventar nombres para las entidades y relaciones cuando están desarrollando un modelo de un sistema grande. Estos nombres deberían ser usados de forma consistente y no deberían entrar en conflicto. El software del diccionario de datos puede comprobar la unicidad de los nombres cuando sea necesario y avisar a los analistas de requerimientos de las duplicaciones de nombres.
2. *Sirve como un almacén de información de la organización.* A medida que el sistema se desarrolla, la información que enlaza el análisis, diseño, implementación y evolución se añade al diccionario de datos, para que toda la información sobre una entidad esté en un mismo lugar.

Las entradas del diccionario de datos mostradas en la Figura 8.9 definen los nombres en el modelo semántico de datos para LIBSYS (Figura 8.8). Se ha simplificado la presentación de este ejemplo obviando algunos nombres y reduciendo la información asociada.

Artículo	Detalle del artículo publicado que puede pedirse por las personas que usen LIBSYS	Entidad	30.12.2002
Autores	Los nombres de los autores del artículo que pueden compartir los honorarios	Atributo	30.12.2002
Comprador	La persona u organización que emite una orden de copia del artículo	Entidad	30.12.2002
Honorarios a pagar a	Una relación 1:1 entre Artículo y la Agencia de derechos de autor a quien se debería abonar el honorario de derechos de autor	Relación	29.12.2002
Dirección (Comprador)	La dirección del comprador. Ésta se utiliza para cualquier información que se requiera sobre la factura en papel	Atributo	31.12.2002

Figura 8.9 Ejemplos de entradas de diccionario de datos.

Todos los nombres del sistema, tanto si son nombres de entidades, relaciones, atributos o servicios, deberían introducirse en el diccionario. El software se utiliza normalmente para crear, mantener y consultar el diccionario. Este software debería integrarse con otras herramientas para que la creación del diccionario se automatice parcialmente. Por ejemplo, las herramientas CASE que soportan el modelado del sistema incluyen soporte para el diccionario de datos e introducen los nombres en el diccionario cuando se utilizan por primera vez en el modelo.

8.4 Modelos de objetos

Una aproximación orientada a objetos para el proceso de desarrollo del software en su totalidad se usa actualmente de forma generalizada, en particular para el desarrollo de sistemas interactivos. Esto significa expresar los requerimientos de los sistemas utilizando un modelo de objetos, diseñar utilizando objetos y desarrollar el sistema en un lenguaje de programación orientado a objetos, como por ejemplo Java o C++.

Los modelos de objetos que usted desarrolla durante el análisis de requerimientos pueden utilizarse para representar tanto los datos del sistema como su procesamiento. A este respecto, dichos modelos combinan algunos de los usos de los modelos de flujo de datos y los modelos semánticos de datos. Los modelos de objetos también son útiles para mostrar cómo se clasifican las entidades en el sistema y se componen de otras entidades.

Para algunas clases del sistema, los modelos de objetos son formas naturales de reflejar las entidades del mundo real que son manipuladas por el sistema. Esto es particularmente cierto cuando el sistema procesa información sobre entidades tangibles, tales como coches, aviones o libros, que tienen atributos claramente identificables. Entidades más abstractas, y de más alto nivel, tales como el concepto de una biblioteca, un sistema de registros médicos o un procesador de texto, son más difíciles de modelar como clases de objetos. Éstos no tienen necesariamente una interfaz sencilla consistente en atributos independientes y operaciones.

El desarrollo de modelos de objetos durante el análisis de requerimientos normalmente simplifica la transición entre el diseño orientado a objetos y la programación. Sin embargo, se observa que los usuarios de un sistema a menudo buscan modelos de objetos no naturales y difíciles de entender. Ellos pueden preferir adoptar un punto de vista más funcional y de pro-

ceso de datos. Por lo tanto, a veces es útil complementar el modelo de objetos con modelos de flujos de datos que muestren el procesamiento de datos en el sistema desde el principio hasta el final.

Una clase de objetos es una abstracción sobre un conjunto de objetos que identifica atributos comunes (como en un modelo semántico de datos) y los servicios u operaciones que son proporcionados por cada objeto. Los objetos son entidades ejecutables que tienen atributos y servicios de la clase de objetos. Los objetos son instanciaciones de la clase de objetos, y pueden crearse muchos objetos a partir de una clase. Generalmente, los modelos desarrollados utilizando análisis se centran en las clases de objetos y en sus relaciones.

En el análisis de requerimientos orientado a objetos, deberían modelarse entidades del mundo real utilizando clases de objetos. No deberían incluirse detalles de los objetos individuales (instanciaciones de la clase) en el sistema. Se pueden crear diferentes tipos de modelos de objetos, mostrando cómo las clases de objeto se relacionan unas con otras, cómo los objetos son agregados para formar otros objetos, cómo los objetos interactúan con otros objetos, etcétera. Cada uno de éstos presenta una información distinta sobre el sistema que se está especificando.

El proceso de análisis para identificar los objetos y las clases de objetos se reconoce como una de las áreas más difíciles del desarrollo orientado a objetos. La identificación de objetos es básicamente la misma para el análisis y para el diseño. Pueden utilizarse los métodos de identificación de objetos tratados en el Capítulo 14, que analiza el diseño orientado a objetos. Aquí se tratan algunos de los modelos de objetos que podrían generarse durante el proceso de análisis.

En los años 90 se propusieron varios métodos de análisis orientados a objetos (Coad y Yourdon, 1990; Rumbaugh *et al.*, 1991; Jacobsen *et al.*, 1993; Booch, 1994). Estos métodos tienen mucho en común, y tres de estos desarrolladores (Booch, Rumbaugh y Jacobsen) decidieron integrar sus aproximaciones para producir un método unificado (Rumbaugh *et al.*, 1999b). El Lenguaje Unificado de Modelado (UML) utilizado en este método unificado se ha convertido en un estándar para el modelado de objetos. UML incluye notaciones para diferentes tipos de modelos de sistemas. Nosotros ya hemos visto los modelos de casos de uso y los diagramas de secuencia en capítulos anteriores y los modelos de máquinas de estados al principio de este capítulo.

Una clase de objetos en UML, como se ha ilustrado en los ejemplos de la Figura 8.10, se representa como un rectángulo orientado verticalmente con tres secciones:

1. El nombre de la clase de objetos está en la sección superior.
2. Los atributos de la clase están en la sección intermedia.
3. Las operaciones asociadas con la clase de objetos están en la sección inferior del rectángulo.

debido a la limitación de espacio para incluir todo sobre UML, aquí sólo se tratan modelos de objetos que muestran cómo los objetos pueden clasificarse y pueden heredar atributos y operaciones de otros objetos, modelos de agregación que muestran cómo están compuestos los objetos, y modelos de comportamiento sencillos, que muestran las interacciones entre los objetos.

8.4.1 Modelos de herencia

El modelado orientado a objetos implica la identificación de clases de objetos que son importantes en el dominio que se está estudiando. Estos objetos se organizan a continuación

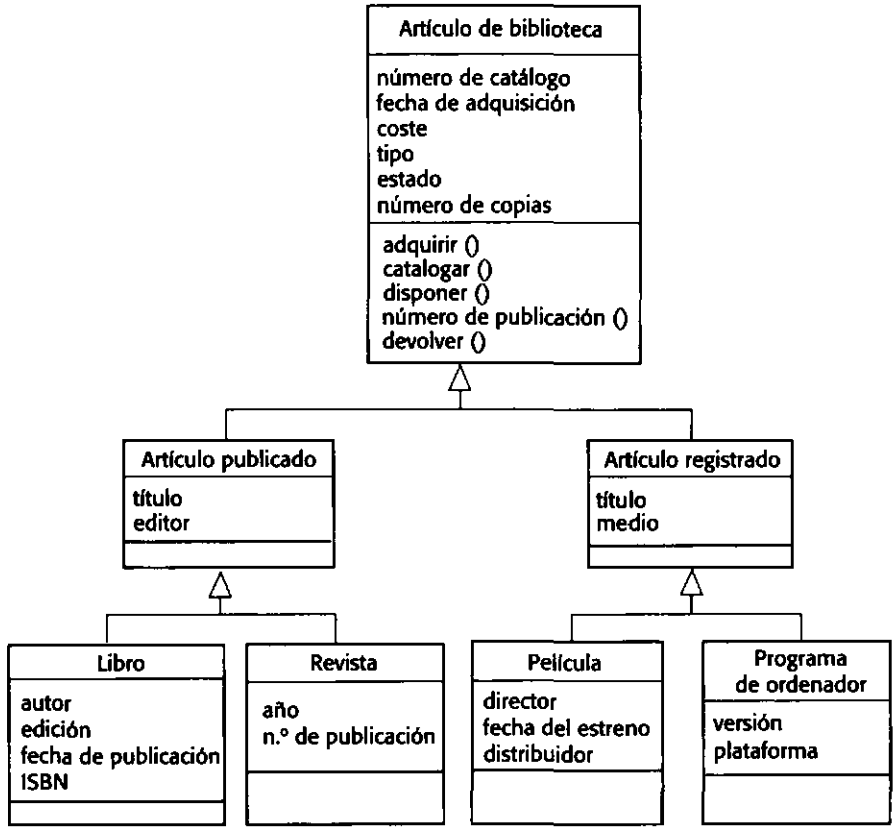


Figura 8.10 Parte de una jerarquía de clases para una biblioteca.

en una taxonomía. Una taxonomía es un esquema de clasificación que muestra cómo una clase de objetos está relacionada con otras clases a través de atributos y servicios comunes.

Para mostrar esta taxonomía, las clases se organizan en una jerarquía de herencia con las clases de objetos más generales al principio de la jerarquía. Los objetos más especializados heredan sus atributos y servicios. Estos objetos especializados pueden tener sus propios atributos y servicios.

La Figura 8.10 ilustra parte de una jerarquía de clases simplificada para un modelo de biblioteca. Esta jerarquía proporciona información sobre los elementos almacenados en la biblioteca. La biblioteca comprende varios elementos, tales como libros, música, grabaciones de películas, revistas y periódicos. En la Figura 8.10, el elemento más general está en la raíz del árbol y tiene un conjunto de atributos y servicios que son comunes a todos los elementos de la biblioteca. Éstos son heredados por las clases **Elemento publicado** y **Elemento registrado**, que añaden sus propios atributos que son heredados a continuación por elementos de niveles inferiores.

La Figura 8.11 es un ejemplo de otra jerarquía de herencia que podría ser parte del modelo de biblioteca. En este caso, se muestran los usuarios de una biblioteca. Hay dos clases de usuarios: aquellos a los que se les permite pedir prestados libros, y aquellos que sólo pueden leer los libros en la biblioteca sin llevárselos.

En la notación UML, la herencia se muestra «hacia arriba» en lugar de «hacia abajo» como en otras notaciones orientadas a objetos o en lenguajes tales como Java, en donde las subclases heredan de las superclases. Es decir, la punta de la flecha (mostrada como

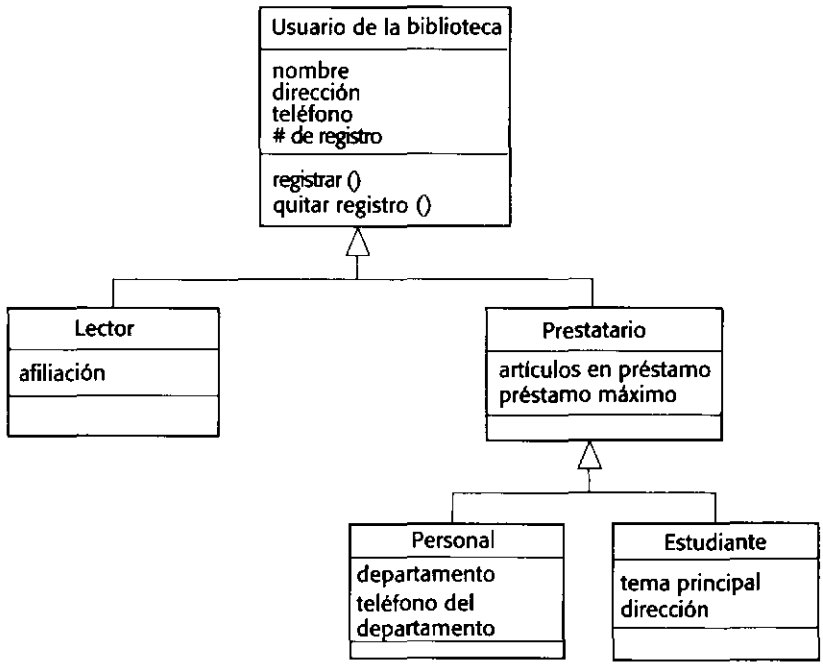


Figura 8.11
Jerarquía de clases de usuarios.

un triángulo) apunta desde las clases que heredan sus atributos y operaciones hasta su superclase. En lugar de utilizar el término *herencia*, UML habla de *relación de generalización*.

El diseño de jerarquías de clases no es fácil, ya que el analista necesita comprender con detalle el dominio en el que el sistema será implantado. Como ejemplo de los problemas sutiles que surgen en la práctica, considere la jerarquía de elementos de la biblioteca. Podría parecer que el atributo **Título** podría situarse como el elemento más general, y a continuación ser heredado por elementos de niveles inferiores.

Sin embargo, si bien cada elemento en una biblioteca debe tener algún tipo de identificador o número de registro, esto no significa que todos los elementos deban tener un título. Por ejemplo, una biblioteca puede almacenar ciertos elementos personales de un político retirado. Muchos de estos elementos, tales como cartas, pueden no tener un título de forma explícita. Éstos serán clasificados utilizando alguna otra clase (no mostrada aquí) que tiene un conjunto diferente de atributos.

Las Figuras 8.10 y 8.11 muestran jerarquías de herencia de clases en las que cada clase de objetos hereda sus atributos y operaciones de una única clase padre. También pueden construirse modelos de herencia en los que una clase tiene varios padres. Sus atributos y servicios son una conjunción de los heredados de cada superclase. La Figura 8.12 muestra un ejemplo de modelo de herencia múltiple que también puede ser parte del modelo de biblioteca.

El problema principal con la herencia múltiple es el diseño de un grafo de herencia en donde los objetos no heredan atributos innecesarios. Entre otros problemas se incluye la dificultad de reorganizar el grafo de herencia cuando se requieren cambios y resolver conflictos de nombres cuando dos o más superclases tienen el mismo nombre pero diferentes significados. A nivel de modelado de sistemas, tales conflictos son relativamente fáciles de resolver alterando manualmente el modelo de objetos. Esto puede ocasionar más problemas en la programación orientada a objetos.

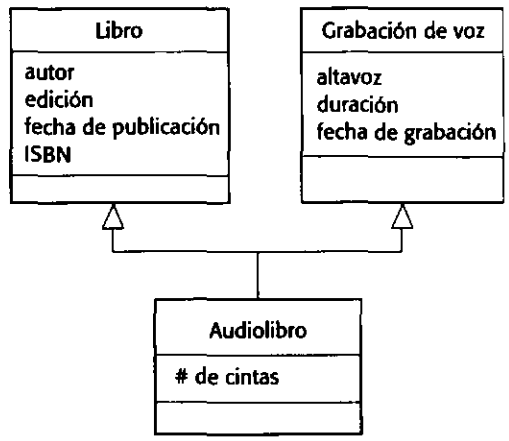


Figura 8.12
Herencia múltiple.

8.4.2 Agregación de objetos

Así como se adquieren atributos y servicios a través de una relación de herencia con otros objetos, algunos objetos son agrupaciones de otros objetos. Es decir, un objeto es un *agregado* de un conjunto de otros objetos. Las clases que representan a estos objetos pueden modelarse utilizando un modelo de agregación, tal y como se muestra en la Figura 8.13. En este ejemplo, se ha modelado un elemento de biblioteca, consistente en un paquete de estudio para un curso universitario. El paquete de estudio comprende apuntes de clase, ejercicios, soluciones ejemplo, copias de las transparencias usadas en las clases y cintas de vídeo.

La notación UML para la agregación consiste en representar la composición incluyendo una figura de diamante colocada sobre el elemento fuente del enlace. Por lo tanto, la Figura 8.13 puede leerse como «Un paquete de estudio está compuesto por uno o varios elementos asignados, paquetes de transparencias OHP, apuntes y cintas de vídeo».

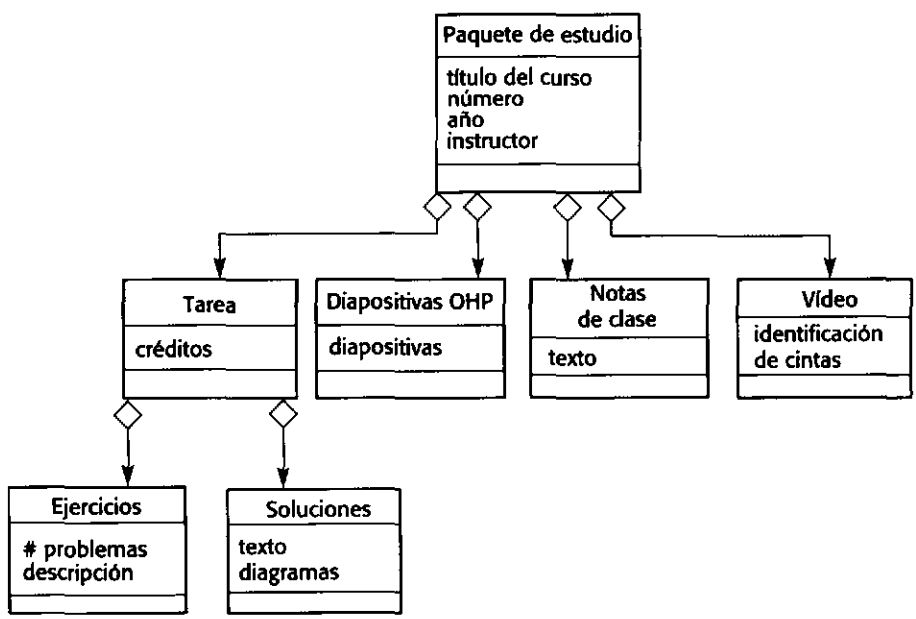


Figura 8.13 Objeto agregado que representa un curso.

8.4.3 Modelado de comportamiento de objetos

Para modelar el comportamiento de los objetos, se tiene que mostrar cómo se utilizan las operaciones proporcionadas por los objetos. En UML, se modelan los comportamientos utilizando escenarios que son representados como casos de uso UML (estudiados en el Capítulo 7). Una forma de modelar los comportamientos es utilizar diagramas de secuencia UML que muestran la secuencia de acciones implicadas en un caso de uso. Además de los diagramas de secuencia, UML también incluye diagramas de colaboración que muestran la secuencia de mensajes intercambiados por los objetos. Estos diagramas son similares a los diagramas de secuencia, por lo que no se tratan aquí.

Se puede ver cómo se pueden utilizar los diagramas de secuencia para modelar el comportamiento en la Figura 8.14, que expande un caso de uso del sistema LIBSYS en el que los usuarios solicitan préstamos a la biblioteca en formato electrónico. Por ejemplo, imagine una situación en la que los paquetes de estudio mostrados en la Figura 8.13 podrían mantenerse en formato electrónico y descargarse a la computadora del estudiante.

En un diagrama de secuencia, los objetos y los actores se alinean en la parte superior del diagrama. Las flechas etiquetadas indican las operaciones; la secuencia de operaciones se lleva a cabo desde arriba hacia abajo. En este escenario, el usuario de la biblioteca accede al catálogo para ver si el elemento solicitado está disponible en formato electrónico; si lo está, el usuario solicita dicho elemento. Por razones de derechos de autor, se debe asignar una licencia al elemento para que exista una transacción entre el elemento y el usuario, que debe aceptar dicha licencia. El elemento solicitado se envía a un objeto servidor de red para su compresión antes de ser enviado al usuario de la biblioteca.

Se puede encontrar otro ejemplo de un diagrama de secuencia que expande un caso de uso de LIBSYS en la Figura 7.8, que muestra la secuencia de actividades implicadas en la impresión de un artículo.

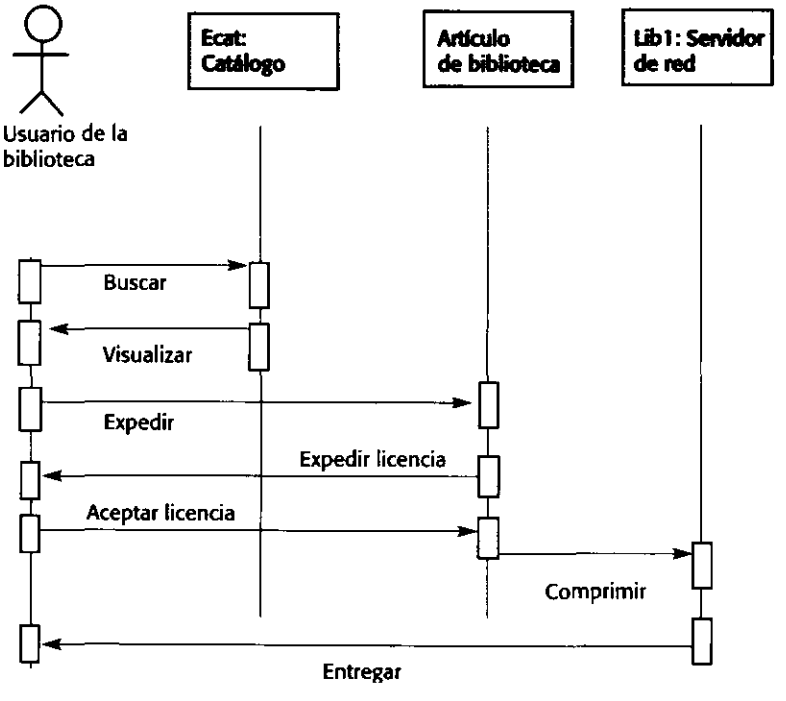


Figura 8.14 Expedición de elementos en formato electrónico.

8.5 Métodos estructurados

Un método estructurado es una forma sistemática de elaborar modelos de un sistema existente o de un sistema que tiene que ser construido. Fueron desarrollados por primera vez en la década de los 70 para soportar el análisis y el diseño del software (Constantine y Yourdon, 1979; Gane y Sarson, 1979; Jackson, 1983) y evolucionaron en las décadas de los 80 y de los 90 para soportar el desarrollo orientado a objetos (Rumbaugh *et al.*, 1991; Robinson, 1992; Jacobsen, *et al.*, 1993; Booch, 1994). Estos métodos orientados a objetos se unieron con la propuesta UML como lenguaje de modelado estándar (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a) y el Proceso Unificado (Rumbaugh *et al.*, 1999b), y más tarde con el Proceso Unificado de Rational (Krutchen, 2000) como un método asociado estructurado. Budgen (Budgen, 2003) resume y compara varios de estos métodos estructurados.

Los métodos estructurados proporcionan un marco para el modelado detallado de sistemas como parte de la elicitación y análisis de requerimientos. La mayoría de métodos estructurados tienen su propio conjunto preferido de modelos de sistemas. Normalmente definen un proceso que puede utilizarse para derivar estos modelos y un conjunto de reglas y guías que aplican a dichos modelos. Se genera una documentación estándar para el sistema. Normalmente se encuentran disponibles herramientas CASE que soportan el uso de métodos. Estas herramientas soportan la edición de modelos y generación de código y documentación, y proporcionan algunas capacidades para comprobar los modelos.

Los métodos estructurados han sido aplicados con éxito en muchos proyectos grandes. Pueden suponer reducciones significativas de coste debido a que utilizan notaciones estándar y aseguran que se produce una documentación de diseño estándar. Sin embargo, los métodos estructurados tienen una serie de inconvenientes:

1. No proporcionan un soporte efectivo para la comprensión o el modelado de requerimientos del sistema no funcionales.
2. No discriminan en tanto que normalmente no incluyen guías que ayuden a los usuarios a decidir si un método es adecuado para un problema concreto. Tampoco incluyen normalmente consejos sobre cómo pueden adaptarse para su uso en un entorno particular.
3. A menudo generan demasiada documentación. La esencia de los requerimientos del sistema puede quedar oculta por el volumen de detalle que se incluye.
4. Los modelos producidos son muy detallados, y los usuarios a menudo los encuentran difíciles de entender. Estos usuarios, por lo tanto, no pueden comprobar el realismo de estos modelos.

En la práctica, sin embargo, los ingenieros de requerimientos y los diseñadores no se restringen a los modelos propuestos por un método determinado. Por ejemplo, los métodos orientados a objetos no sugieren normalmente que los modelos de flujos de datos deban desarrollarse. Sin embargo, según mi experiencia, dichos modelos son a menudo útiles como parte del proceso de análisis de requerimientos debido a que pueden presentar un panorama general del procesamiento en el sistema desde el principio hasta el final. También pueden contribuir directamente a la identificación de los objetos (los datos que fluyen) y la identificación de las operaciones sobre esos objetos (las transformaciones).

Las herramientas CASE de análisis y diseño soportan la creación, edición y análisis de las notaciones gráficas utilizadas en los métodos estructurados. La Figura 8.15 muestra los componentes que pueden ser incluidos en los entornos que soportan métodos.

Figura 8.15 Los componentes de una herramienta CASE para el soporte de métodos estructurados.

Las herramientas que soportan métodos completos, tal y como se ilustra en la Figura 8.15, normalmente incluyen:

1. *Editores de diagramas* utilizados para crear modelos de objetos, modelos de datos, modelos de comportamiento, etcétera. Estos editores no son simplemente herramientas de dibujo puesto que identifican los tipos de entidades en el diagrama. Captan la información sobre estas entidades y guardan esta información en el repositorio central.
2. *Herramientas de análisis y comprobación de diseños* que procesan el diseño e informan sobre errores y anomalías. Pueden integrarse con el sistema de edición para que los errores del usuario sean detectados en etapas tempranas del proceso de desarrollo.
3. *Lenguajes de consulta del repositorio* que permite al diseñador encontrar diseños e información asociada a los diseños en el repositorio.
4. *Un diccionario de datos* que mantiene información sobre las entidades utilizadas en el diseño de un sistema.
5. *Herramientas de generación y definición de informes* que obtienen información del almacén central y generan automáticamente la documentación del sistema.
6. *Herramientas de definición de formularios* que permiten especificar los formatos de pantallas y de documentos.
7. *Facilidades para importar/exportar* que permiten el intercambio de información desde el repositorio central con otras herramientas de desarrollo.
8. *Generadores de código* que generan código o esqueletos de código de forma automática a partir del diseño capturado en el almacén central.

La mayoría de los conjuntos de herramientas CASE completos permiten al usuario generar un programa o un fragmento de un programa a partir de la información proporcionada en el modelo del sistema. Las herramientas CASE a menudo soportan diferentes lenguajes para que el usuario pueda generar un programa en C, C++ o Java a partir del mismo modelo de diseño. Debido a que los modelos excluyen detalles de bajo nivel, el generador de código en un banco de trabajo de diseño no puede normalmente generar el sistema completo. Normalmente es necesaria alguna codificación manual para añadir detalles al código generado.

PUNTOS CLAVE

- Un modelo es una vista abstracta de un sistema que prescinde de algunos detalles del mismo. Pueden desarrollarse modelos del sistema complementarios para presentar otra información sobre dicho sistema.
- Los modelos de contexto muestran cómo el sistema que se está modelando se ubica en un entorno con otros sistemas y procesos. Definen los límites del sistema. Los modelos arquitectónicos, los modelos de procesos y modelos de flujos de datos pueden utilizarse como modelos de contexto.
- Los diagramas de flujo de datos pueden utilizarse para modelar el procesamiento de los datos llevado a cabo por un sistema. El sistema se modela como un conjunto de transformaciones de datos con funciones que actúan sobre los datos.
- Los modelos de máquina de estados se utilizan para modelar el comportamiento del sistema en respuesta a eventos internos o externos.
- Los modelos semánticos de datos describen la estructura lógica de los datos importados y exportados por el sistema. Estos modelos muestran las entidades del sistema, sus atributos y las relaciones en las que intervienen.
- Los modelos de objetos describen las entidades lógicas del sistema y su clasificación y agregación. Combinan un modelo de datos con un modelo de procesamiento. Posibles modelos de objetos que pueden desarrollarse incluyen modelos de herencia, modelos de agregación y modelos de comportamiento.
- Los modelos de secuencia que muestran las interacciones entre actores y objetos en un sistema se utilizan para modelar el comportamiento dinámico.
- Los métodos estructurados proporcionan un marco para soportar el desarrollo de modelos del sistema. Los métodos estructurados normalmente son soportados de forma extensiva por herramientas CASE, incluyendo la edición de modelos y la comprobación y generación de código.

LECTURAS ADICIONALES

Software Design, 2nd ed. Si bien este libro se centra fundamentalmente en el diseño del software, el autor analiza varios métodos estructurados que también pueden utilizarse en el proceso de ingeniería de requerimientos. No se centra solamente en aproximaciones orientadas a objetos. (D. Budgen, 2003, Addison-Wesley.)

Requirements Analysis and System Design. Este libro se centra en el análisis de sistemas de información y explica cómo pueden utilizarse diferentes modelos UML en el proceso de análisis. (L. Maciaszek, 2001, Addison-Wesley.)

Software Engineering with Objects and Components. Una breve introducción fácil de leer para el uso de UML en la especificación y diseño del sistema. Aunque no contiene una completa descripción de la notación UML, este libro indudablemente es el mejor si usted está intentando aprender y comprender dicha notación. (P. Stevens with R. Pooley, 1999, Addison-Wesley.)

EJERCICIOS

8.1 Dibuje un modelo de contexto para un sistema de información de pacientes en un hospital. Usted puede hacer cualquier suposición razonable sobre otros sistemas existentes en el hospital, pero su modelo debe incluir un sistema de admisión de pacientes y un sistema de almacenamiento de imágenes de rayos X así como otros almacenamientos de diagnósticos.

8.2 Basándose en su experiencia con los cajeros automáticos, dibuje un diagrama de flujo de datos para modelar el procesamiento de los datos que se debe realizar cuando un cliente solicita efectivo en un cajero.

8.3 Modele el procesamiento de los datos que podría tener lugar en un sistema de correo electrónico. Usted debería modelar el proceso de envío de correos y de recepción de correos de forma separada.

8.4 Dibuje una máquina de estados que modele el software de control para:

- Una lavadora automática que tenga diferentes programas para distintos tipos de ropa.
- El software para un reproductor de DVD.
- Un sistema de contestador telefónico que almacene mensajes entrantes y visualice el número de mensajes aceptados en un LED. El sistema debería permitir al propietario del teléfono realizar llamadas desde cualquier lugar al contestador, teclear una secuencia de números (identificados por tonos) y reproducir los mensajes almacenados.

8.5 Un modelo de sistema de software puede representarse como un grafo dirigido en el que los nodos son las entidades en el modelo y los arcos son las relaciones entre esas entidades. Las entidades y las relaciones en el modelo pueden ser etiquetadas con un nombre y otra información. Cada entidad en el modelo tiene un tipo asociado y puede «expandirse» en un submodelo. Dibuje un modelo de datos que describa la estructura de un modelo de sistema de software.

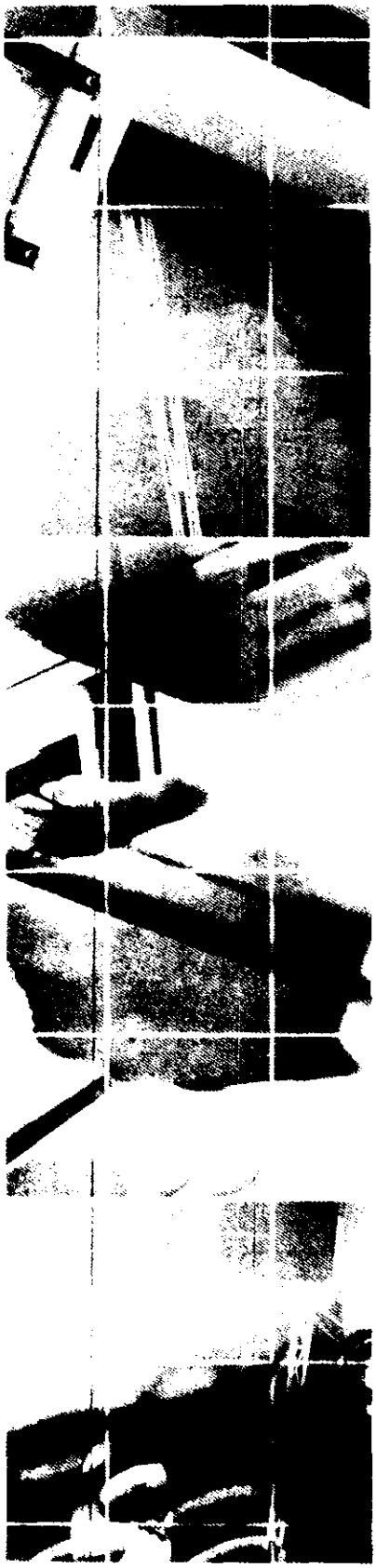
8.6 Modele las clases de objetos que podrían utilizarse en un sistema de correo electrónico. Si ha intentado realizar el Ejercicio 8.3, describa las similitudes y diferencias entre el modelo de procesamiento de datos y el modelo de objetos.

8.7 Utilizando la información sobre los datos del sistema mostrados en la Figura 8.8, dibuje un diagrama de secuencia que muestre una posible secuencia de acciones que tenga lugar cuando un nuevo artículo es catalogado por el sistema LIBSYS.

8.8 Desarrolle un modelo de objetos, incluyendo un diagrama de jerarquía de clases y un diagrama de agregación que muestre los principales componentes de un sistema de computadora personal y su software de sistema.

8.9 Desarrolle un diagrama de secuencia que muestre las interacciones implicadas cuando un estudiante se registra para un curso en una universidad. Los cursos pueden tener limitadas las plazas, de forma que el proceso de registro debe incluir la comprobación de que hay plazas disponibles. Suponga que los estudiantes acceden a un catálogo electrónico de cursos para encontrar cursos disponibles.

8.10 ¿En qué circunstancias no recomendaría el uso de métodos estructurados para el desarrollo de sistemas?



9 Especificación de sistemas críticos

Objetivos

El objetivo de este capítulo es explicar cómo especificar requerimientos de confiabilidad funcionales y no funcionales para sistemas críticos. Cuando haya leído este capítulo:

- comprenderá cómo los requerimientos de confiabilidad para sistemas críticos pueden identificarse mediante el análisis de riesgos con los que se enfrentan dichos sistemas;
- comprenderá que los requerimientos de seguridad se obtienen a partir del análisis de riesgos del sistema en vez de mediante personal externo al sistema;
- comprenderá el proceso de obtención de requerimientos de protección y cómo dichos requerimientos son utilizados para combatir diferentes tipos de amenazas al sistema;
- comprenderá las métricas para la especificación de la fiabilidad y cómo estas métricas pueden usarse para especificar requerimientos de fiabilidad.

Contenidos

En septiembre de 1993, un avión aterrizó en el aeropuerto de Warsaw en Polonia durante una fuerte tormenta. Durante nueve segundos después del aterrizaje, los frenos del sistema de frenado controlado por computadora no funcionaron. El avión sobrepasó el final de la pista de aterrizaje, chocó contra un montículo de tierra y se incendió. La subsiguiente investigación demostró que el software del sistema de frenos funcionó perfectamente de acuerdo con su especificación. Pero, por razones en las que no procede entrar aquí, el sistema de frenos no reconoció que el avión había aterrizado. Una característica de seguridad del avión detuvo el despliegue del sistema de frenado, ya que esto puede ser peligroso si el avión está en el aire. El fallo de funcionamiento del sistema se debió a un error en la especificación del sistema.

Esto ilustra la importancia de la especificación de sistemas críticos. Debido a los altos costes potenciales de un fallo de funcionamiento del sistema, es importante asegurar que la especificación de los sistemas críticos refleja con exactitud las necesidades reales de los usuarios del sistema. Si no se consigue la especificación correcta, entonces, independientemente de la calidad del desarrollo del software, el sistema no será confiable.

La necesidad de confiabilidad en los sistemas críticos genera requerimientos funcionales y no funcionales del sistema:

1. Los requerimientos funcionales del sistema se generan para definir la comprobación de errores y facilidades de recuperación y características que proporcionan protección frente a fallos de funcionamiento del sistema.
2. Los requerimientos no funcionales se generan para definir la fiabilidad y disponibilidad requeridas por el sistema.

Además de estos requerimientos, las consideraciones sobre seguridad y protección pueden generar otro tipo de requerimiento que es difícil de clasificar como funcional o no funcional. Son requerimientos de alto nivel que quizás se describen mejor como requerimientos «no debería». En contraste con los requerimientos funcionales normales que definen lo que el sistema debería hacer, los requerimientos «no debería» definen el comportamiento del sistema que no es aceptable. Ejemplos de requerimientos «no debería» son los siguientes:

- El sistema no debería permitir a los usuarios modificar los permisos de acceso de cualquier fichero que no hayan creados ellos mismos (protección).
- El sistema no debería permitir seleccionar el modo de propulsión inversa cuando el avión esté en el aire (seguridad).
- El sistema no debería permitir la activación simultánea de más de tres señales de alarma (seguridad).

Algunas veces estos requerimientos «no debería» se descomponen en requerimientos funcionales de software más específicos. De forma alternativa, las decisiones de implementación pueden posponerse hasta que se diseñe el sistema.

Los requerimientos de usuario para sistemas críticos siempre se especifican utilizando el lenguaje natural y modelos de sistemas. Sin embargo, como se indica en el Capítulo 10, la especificación formal y la verificación asociada son probablemente las de mayor coste efectivo en el desarrollo de sistemas críticos (Hall, 1996; Hall y Chapman, 2002; Wordsworth, 1996). Las especificaciones formales no son solamente una base para la verificación del diseño y la implementación. Son la forma más precisa de especificar sistemas y, por lo tanto, reducen los malentendidos. Además, la construcción de una especificación formal fuerza a un análisis detallado de los requerimientos, lo que es una forma efectiva de descubrir problemas en la especificación. En una especificación en lenguaje natural, los errores pueden encubrirse por la imprecisión del lenguaje. Éste no es el caso si el sistema se especifica formalmente.

9.1 Especificación dirigida por riesgos

La especificación de sistemas críticos no es una alternativa al proceso habitual de especificación de requerimientos. En su lugar, complementa a este último proceso haciendo hincapié en la confiabilidad del sistema. Su objetivo es comprender los riesgos con los que se enfrenta el sistema y generar requerimientos de confiabilidad para tratar dichos riesgos. La especificación dirigida por riesgos es una aproximación que ha sido ampliamente utilizada por los desarrolladores de sistemas de seguridad críticos y sistemas de protección críticos. En sistemas de seguridad críticos, los riesgos son contingencias que pueden provocar accidentes; en sistemas de protección críticos, los riesgos son vulnerabilidades que pueden conducir a un ataque con éxito al sistema. Sin embargo, una aproximación dirigida por riesgos es aplicable a cualquier sistema en el que la confiabilidad sea un atributo crítico.

El proceso de especificación dirigido por riesgos implica comprender los riesgos con los que se enfrenta el sistema, descubriendo sus causas fundamentales y generando los requerimientos para gestionar dichos riesgos. La Figura 9.1 muestra el proceso iterativo del análisis de riesgos:

1. *Identificación de riesgos*. Se identifican los riesgos potenciales que podrían surgir. Éstos dependen del entorno en el que se va a utilizar el sistema.
2. *Análisis y clasificación de riesgos*. Los riesgos se consideran de forma independiente. Aquellos que son potencialmente serios y no previsibles se seleccionan para un análisis posterior. En esta etapa, algunos riesgos pueden eliminarse simplemente debido a que es muy improbable que surjan (por ejemplo, tormentas eléctricas y terremotos).
3. *Descomposición de riesgos*. Cada riesgo se analiza individualmente para descubrir las causas potenciales fundamentales de ese riesgo. Se pueden utilizar técnicas tales como el análisis del árbol de defectos (comentado más adelante en este capítulo).
4. *Valoración de la reducción de riesgos*. Se hacen propuestas sobre las formas en las que los riesgos identificados pueden reducirse o eliminarse. Éstos generan entonces requerimientos de confiabilidad del sistema que definen las defensas frente al riesgo y cómo el riesgo va a ser gestionado si ocurre.

Para sistemas grandes, el análisis de riesgos se puede estructurar en fases. El análisis de riesgos multifase es necesario para sistemas grandes tales como plantas químicas o aviones. Las fases del análisis de riesgos comprenden:

- Análisis preliminar de riesgos en el que se identifican los riesgos principales.
- Análisis más detallado de riesgos del sistema y subsistemas.
- Análisis de riesgos del software en el que se consideran los riesgos de fallos de funcionamiento del software.
- Análisis de riesgos operacionales que comprenden la interfaz de usuario del sistema y los riesgos que surgen por los errores del operador.

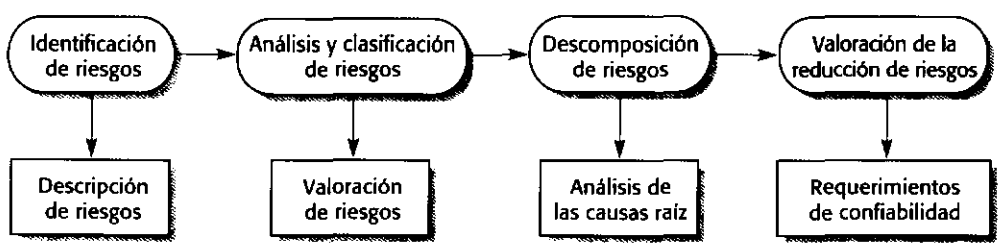


Figura 9.1 Especificación dirigida por riesgos.

Leveson (Leveson, 1995) trata el proceso multifase de análisis de riesgos en su libro sobre sistemas de seguridad críticos.

9.1.1 Identificación de riesgos

El objetivo de la identificación de riesgos, la primera etapa del proceso de análisis de riesgos, es identificar los riesgos con los que el sistema tiene que tratar. Esto puede ser un proceso difícil y complejo debido a que en ocasiones los riesgos aparecen como consecuencia de interacciones entre el sistema y condiciones inesperadas del entorno. El accidente de Warsaw que se ha comentado al principio ocurrió cuando los vientos de costado generados durante una tormenta hicieron que el avión se inclinara de forma que aterrizó con una rueda en vez de con las dos.

En sistemas de seguridad críticos, los riesgos principales son contingencias que provocan accidentes. Se puede abordar el problema de identificación de contingencias considerando las diferentes clases de contingencias, tales como contingencias físicas, eléctricas, biológicas, radiactivas, de fallo de servicio, etc. Cada una de estas clases puede analizarse para descubrir las contingencias asociadas. También deben identificarse las posibles combinaciones de contingencias.

En el Capítulo 3 se presentó un ejemplo de un sistema de una bomba de insulina. Al igual que muchos dispositivos médicos, éste es un sistema de seguridad crítico. Algunas de las contingencias o riesgos que podrían surgir en este sistema son:

1. Sobredosis de insulina (fallo de servicio).
2. Dosis baja de insulina (fallo de servicio).
3. Fallo de energía debido a batería agotada (eléctrico).
4. Interferencia eléctrica con otros equipamientos médicos tal como un marcapasos (eléctrico).
5. Contacto inadecuado entre el sensor y el actuador causado por una mala conexión (física).
6. Las partes de la máquina sobre el cuerpo del paciente interrumpen su funcionamiento (física).
7. Infección provocada por la introducción de una máquina (biológica).
8. Reacción alérgica a los materiales o insulina usada en la máquina (biológica).

Los riesgos relacionados con el software normalmente se asocian a fallos en la ejecución de un servicio del sistema o con el fallo de sistemas de monitorización y protección. Los sistemas de monitorización pueden detectar condiciones potenciales de contingencia tales como fallos en el suministro eléctrico.

Los ingenieros con experiencia, que trabajan con expertos en el dominio y asesores de seguridad profesionales, deberían identificar los riesgos del sistema. Las técnicas de trabajo en grupo como la tormenta de ideas (*brainstorming*) se pueden usar para identificar los riesgos. Los analistas con experiencia directa en incidentes previos también pueden ser capaces de identificar los riesgos.

9.1.2 Análisis y clasificación de riesgos

El proceso de análisis y clasificación de riesgos trata principalmente de comprender la probabilidad de que surja un riesgo y las consecuencias potenciales que tendrían lugar si se pro-

duce un accidente o incidente asociado a ese riesgo. Necesitamos realizar este análisis para comprender si un riesgo es una amenaza seria para el sistema o el entorno y proporcionar una base para decidir los recursos que se deberían usar para gestionar el riesgo.

Para cada riesgo, el resultado del proceso de análisis y clasificación de riesgos es una decisión de aceptación. Los riesgos pueden clasificarse de tres formas:

1. *Intolerable*. El sistema debe diseñarse de tal forma que, o bien el riesgo no pueda ocurrir, o si ocurre, que no provoque un accidente. Los riesgos intolerables deberían ser, normalmente, aquellos que amenacen la vida humana o la estabilidad financiera de una empresa y que tienen una probabilidad de ocurrencia significativa. Un ejemplo de un riesgo intolerable para un sistema de comercio electrónico en una librería a través de Internet, por ejemplo, sería el riesgo de que el sistema cayese durante más de un día.
2. *Tan bajo como razonablemente práctico (ALARP)*. El sistema debe diseñarse para que la probabilidad de que ocurra un accidente debido a la contingencia correspondiente se minimice, sujeto a otras consideraciones tales como coste y entrega. Los riesgos clasificados como ALARP son aquellos que tienen menos consecuencias serias o con menos probabilidad de ocurrir. Un riesgo ALARP para un sistema de comercio electrónico podría ser la corrupción de las imágenes de una página web que muestra el logotipo de la empresa. Esto no es deseable comercialmente, pero es improbable que tenga consecuencias serias a corto plazo.
3. *Aceptable*. Si bien los diseñadores del sistema deberían realizar todos los posibles pasos para reducir la probabilidad de ocurrencia de una contingencia «aceptable», dichos pasos no deberían incrementar los costes, tiempo de entrega u otros atributos no funcionales del sistema. Un ejemplo de un riesgo aceptable para un sistema de comercio electrónico es el riesgo de que la gente use navegadores web con versiones beta que podrían no completar con éxito los pedidos.

La Figura 9.2 (Brazendale y Bell, 1994), desarrollada para sistemas de seguridad críticos, muestra estas tres regiones. La parte sombreada del diagrama refleja los costes para asegurar que los riesgos no provocan accidentes o incidentes. El coste de diseño del sistema para tener en cuenta el riesgo es una función de la anchura del triángulo. Los costes más altos se dan con los riesgos de la parte superior del diagrama; los más bajos, con los riesgos de la cima del triángulo.

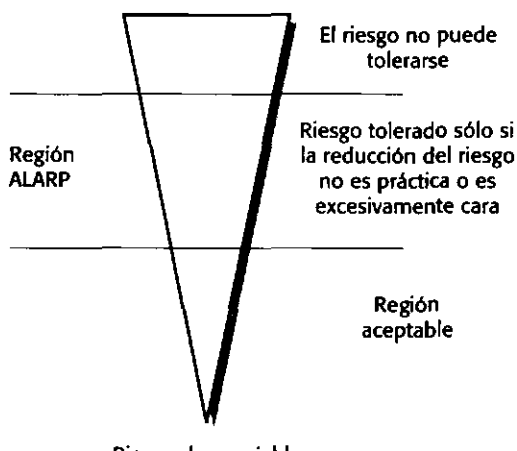


Figura 9.2 Niveles de riesgo.

Los límites entre las regiones de la Figura 9.2 tienden a moverse a lo largo del tiempo, debido a expectativas públicas de seguridad y a consideraciones políticas. Si bien los costes financieros de aceptar los riesgos y de pagar cualquier accidente que pueda tener lugar pueden ser menores que los costes de prevención de accidentes, la opinión pública puede demandar que dichos costes adicionales sean aceptados. Por ejemplo, puede ser más barato para una empresa tratar el problema de la polución en las raras ocasiones en las que ocurra, que instalar sistemas para prevenir la polución. Esto hubiera sido aceptable en los años 60 y 70, pero probablemente hoy en día no es aceptable pública o políticamente. En este caso, el límite entre la región intolerable y la región ALARP se ha movido hacia abajo para que los riesgos que hubieran sido aceptados en el pasado sean ahora intolerables.

La valoración del riesgo comprende la estimación de la probabilidad y severidad del riesgo. Normalmente éste es muy difícil de llevar a cabo de forma exacta y generalmente depende de valoraciones de ingeniería. Las probabilidades y severidades de los riesgos se asignan utilizando términos relativos tales como *probable, improbable,* y *raro* y *alto, medio* y *bajo*. La experiencia previa en el sistema puede permitir asociar algún valor numérico a estos términos. Sin embargo, debido a que los accidentes son relativamente poco comunes, es muy difícil validar la exactitud de este valor.

La Figura 9.3 muestra una clasificación de riesgos para los riesgos (contingencias) identificados en la sección previa para el sistema de suministro de insulina. Las estimaciones son simplemente ilustrativas. La figura no muestra necesariamente las probabilidades y severidades reales que podrían tener lugar en un análisis real de un sistema de suministro de insulina. Se debe hacer notar que a corto plazo una sobredosis de insulina es potencialmente más seria que una dosis baja de insulina. Una dosis baja de insulina puede provocar pérdida de conocimiento, estado de coma y en última instancia la muerte.

Las contingencias 3-8 no están relacionadas con el software; por lo tanto, no se hablará más de ellas aquí. Para contabilizar estas contingencias, la máquina debería tener software de autoverificación para monitorizar el estado del sistema y avisar de algunas de estas contingencias. A menudo la advertencia permitirá detectar la contingencia antes de que provoque un accidente. Ejemplos de contingencias que podrían detectarse son fallos de energía eléctrica y ubicación incorrecta de la máquina. Por supuesto, la monitorización del software está relacionada con la seguridad ya que un fallo en la detección de una contingencia podría provocar un accidente.

1. Sobredosis de insulina	Media	Alta	Alto	Intolerable
2. Dosis baja de insulina	Media	Baja	Bajo	Aceptable
3. Fallo de energía eléctrica	Alta	Baja	Bajo	Aceptable
4. Máquina ajustada incorrectamente	Alta	Alta	Alto	Intolerable
5. La máquina deja de funcionar	Baja	Alta	Medio	ALARP
6. La máquina provoca una infección	Media	Media	Medio	ALARP
7. Interferencia eléctrica	Baja	Alta	Medio	ALARP
8. Reacción alérgica	Baja	Baja	Bajo	Aceptable

Figura 9.3 Análisis de riesgos de contingencias identificadas en una bomba de insulina.

9.1.3 Descomposición de riesgos

La descomposición de riegos es el proceso de descubrir las causas fundamentales de los riesgos de un sistema particular. Las técnicas para la descomposición de riesgos se han derivado principalmente del desarrollo de sistemas de seguridad críticos en los que el análisis de contingencias es una parte central del proceso de seguridad. El análisis de riesgos puede ser deductivo o inductivo. Las técnicas deductivas o técnicas descendentes, cuyo uso tiende a ser más fácil, comienzan con el riesgo y trabajan a partir de él hasta llegar al posible fallo de funcionamiento del sistema; las técnicas inductivas o técnicas ascendentes parten de una propuesta de fallo de funcionamiento del sistema e identifican qué contingencias podrían ocurrir que diesen lugar a dicho fallo.

Se han propuesto varias técnicas como posibles aproximaciones a la descomposición de riesgos. Éstas incluyen revisiones y listas de comprobación, así como técnicas más formales como el análisis de redes de Petri (Peterson, 1981), lógica formal (Jahanian y Mok, 1986) y análisis de árbol de defectos (Leveson y Stolzy, 1987; Storey, 1996).

Aquí se trata el análisis del árbol de defectos. Esta técnica fue desarrollada para sistemas de seguridad críticos y es relativamente fácil de comprender sin un conocimiento especial del dominio. El análisis del árbol de defectos implica identificar eventos no deseados y trabajar hacia atrás a partir de ese evento para descubrir las posibles causas de la contingencia. Se debe poner la contingencia como raíz del árbol e identificar los estados que pueden conducir a dicha contingencia. A continuación, para cada uno de esos estados, se identifican los estados que pueden conducir hasta él y se continúa esta descomposición hasta que se identifiquen las causas raíz del riesgo. Los estados se pueden unir mediante símbolos «and» y «or». Los riesgos que requieren una combinación de causas raíz normalmente son menos probables que los riesgos que pueden resultar de una única causa raíz.

La Figura 9.4 es el árbol de defectos para las contingencias relacionadas con el software del sistema de suministro de insulina. Las dosis bajas y las sobredosis de insulina realmente representan una sola contingencia, denominada «dosis incorrecta de insulina suministrada», a partir de la cual se puede dibujar un único árbol de defectos. Por supuesto, cuando se especifica cómo debería reaccionar el software ante las contingencias, se debe distinguir entre una dosis baja de insulina y una sobredosis de insulina.

El árbol de defectos de la Figura 9.4 está incompleto. Únicamente se han descompuesto por completo los defectos potenciales del software. No se muestran los defectos del hardware tales como un bajo nivel de batería causante de un fallo en el sensor. En este nivel, no es posible hacer un análisis más detallado. Sin embargo, si se incluye más nivel de detalle en el diseño y la implementación, se pueden realizar árboles de defectos más detallados. Leveson y Harvey (Leveson y Harvey, 1983) y Leveson (Leveson, 1985) muestran cómo los árboles de defectos pueden generarse durante el diseño del software descendiendo hasta el nivel de sentencias individuales del lenguaje de programación.

Los árboles de defectos se usan también para identificar problemas hardware potenciales. Un árbol de defectos puede proporcionar un mayor entendimiento de los requerimientos del software para detectar y, quizás, corregir estos problemas. Por ejemplo, las dosis de insulina no se administran con una frecuencia muy alta, no más de dos o tres veces por hora y algunas veces con menos frecuencia. Por ello, la capacidad del procesador está disponible para ejecutar programas de autocomprobación y diagnosis. Los errores hardware tales como errores de sensores, de la bomba o del temporizador pueden descubrirse, además de mostrar los correspondientes avisos, antes de que dichos errores tengan consecuencias serias sobre el paciente.

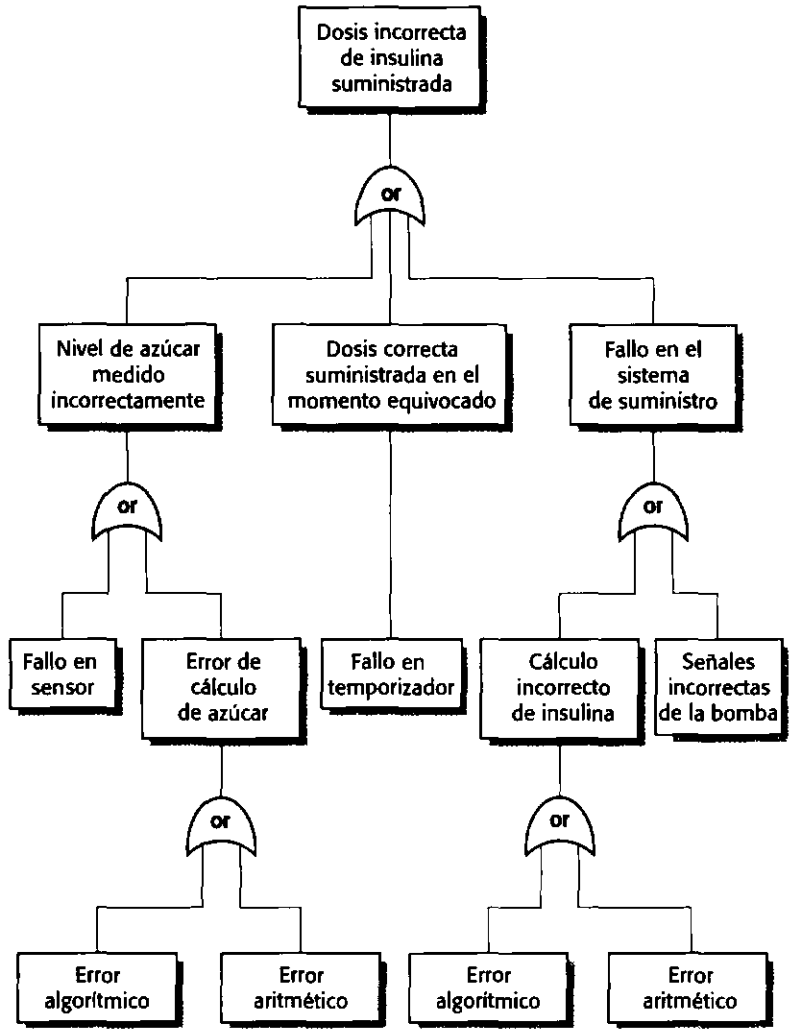


Figura 9.4 Árbol de defectos para un sistema de suministro de insulina.

9.1.4 Valoración de la reducción de riesgos

Una vez que se han identificado los riesgos potenciales y sus causas, se deberían obtener los requerimientos de confiabilidad del sistema que gestionan los riesgos y aseguran que no ocurran incidentes ni accidentes. Hay tres posibles estrategias que pueden usarse:

1. *Evitación de riesgos*. El sistema se diseña para que el riesgo o la contingencia no pueda ocurrir.
2. *Detección y eliminación de riesgos*. El sistema se diseña para que los riesgos se detecten y neutralicen antes de que provoquen un accidente.
3. *Limitación del daño*. El sistema se diseña para que las consecuencias de un accidente se minimicen.

Normalmente, los diseñadores de sistemas críticos utilizan una combinación de estas aproximaciones. En un sistema de seguridad crítico, las contingencias intolerables pueden manejarse minimizando su probabilidad y añadiendo un sistema de protección que proporcione copias de seguridad. Por ejemplo, en un sistema de control de una planta química, el sistema

intentará detectar y evitar el exceso de presión en el reactor. Sin embargo, también debería haber un sistema de protección independiente que monitorice la presión y abra una válvula de escape si se detecta una presión elevada.

En el sistema de suministro de insulina, un «estado seguro» es un estado de apagado en el que no se suministra insulina. Durante un corto periodo esto no supondrá una amenaza para la salud del diabético. Si se consideran los problemas potenciales del software identificados en la Figura 9.4, se podrían dar las siguientes «soluciones»:

1. *Error aritmético.* Tiene lugar cuando algún cálculo aritmético provoca un error de representación. La especificación debe identificar todos los posibles errores aritméticos que puedan ocurrir. Éstos dependen del algoritmo utilizado. La especificación debería establecer que se incluya un manejador de excepciones para cada error aritmético identificado. La especificación debería determinar la acción a realizar para cada uno de estos errores en caso de que aparezcan. Una acción segura es apagar el sistema de suministro y activar una alarma de advertencia.
2. *Error algorítmico.* Ésta es una situación más difícil ya que no es posible detectar una anomalía concreta. Podría detectarse comparando la dosis calculada de insulina requerida con la dosis previamente suministrada. Si ésta es mucho mayor, esto puede significar que la cantidad se ha calculado de forma incorrecta. El sistema también puede registrar las secuencias de dosis. Después de un número de dosis entregadas por encima de la media, se debe emitir una advertencia y limitar las nuevas dosis.

Algunos de los requerimientos de seguridad resultantes para el sistema de suministro de insulina se muestran en la Figura 9.5. Éstos son requerimientos del usuario y, naturalmente, deberían expresarse con más detalle en una especificación final del sistema. En estos requerimientos las referencias a las Tablas 3 y 4 hacen relación a tablas que se deberían incluir en el documento de requerimientos.

9.2 Especificación de la seguridad

El proceso de gestión de riesgos comentado hasta ahora ha evolucionado a partir de los procesos desarrollados para sistemas de seguridad críticos. Hasta hace relativamente poco tiempo, los sistemas de seguridad críticos eran mayormente sistemas de control en los que los fa-

SR1:	**El sistema no deberá suministrar a un usuario del sistema una dosis de insulina mayor que una dosis máxima especificada.**
SR2:	**El sistema no deberá suministrar a un usuario del sistema una dosis acumulada diaria de insulina mayor que una dosis máxima especificada.**
SR3:	**El sistema deberá incluir una facilidad para diagnóstico de hardware que deberá ejecutarse al menos cuatro veces por hora.**
SR4:	**El sistema deberá incluir un manejador de excepciones para todas las excepciones identificadas en la Tabla 3.**
SR5:	**Una alarma audible debería sonar cuando se descubra cualquier anomalía hardware o software y también debería visualizar un mensaje de diagnóstico como el definido en la Tabla 4.**
SR6:	**Cuando se genere un evento que provoque una alarma en el sistema, el sistema de insulina debería suspenderse hasta que el usuario haya reinicializado el sistema y eliminado la alarma.**

Figura 9.5 Ejemplos de requerimientos de seguridad para una bomba de insulina.

llos del equipamiento que se estaba controlando podían causar lesiones. En las décadas de los 80 y 90, a medida que los sistemas de control se utilizan de forma más extensa, la comunidad de ingenieros de seguridad desarrolló estándares para la especificación y desarrollo de sistemas de seguridad críticos.

El proceso de especificación y garantía de seguridad es parte de un ciclo de vida completo de seguridad definido en un estándar internacional para la gestión de seguridad IEC 61508 (IEC, 1998). Este estándar se desarrolló específicamente para la protección de sistemas tales como un sistema de parada de un tren si éste se salta una señal en rojo. Si bien el estándar mencionado puede usarse para sistemas de seguridad críticos más generales, como sistemas de control, su separación de una especificación segura a partir de una especificación del sistema más general parece inapropiada para sistemas de información críticos. La Figura 9.6 ilustra el modelo de sistema que es asumido por el estándar IEC 61508.

La Figura 9.7 es una simplificación de la presentación de Redmill del ciclo de vida de seguridad (Redmill, 1998). Como puede verse en la Figura 9.7, este estándar abarca todos los aspectos de la gestión de la seguridad desde el ámbito inicial de la definición pasando por la planificación y desarrollo del sistema hasta la desmantelación del mismo.

En este modelo, el sistema de control controla dispositivos que tienen asociados unos requerimientos de seguridad de nivel alto. Estos requerimientos de nivel alto generan dos tipos de requerimientos de seguridad más detallados que se aplican a los dispositivos para la protección del sistema:

1. *Requerimientos de seguridad funcionales* que definen las funciones de seguridad del sistema.
2. *Requerimientos de integridad seguros* que definen la fiabilidad y disponibilidad de la protección del sistema. Dichos requerimientos están basados en la forma esperada de uso del sistema de protección y pretenden asegurar que dicho sistema funcionará cuando sea necesario. Los sistemas se clasifican según un nivel de integridad segura (SIL) desde 1 hasta 4. Cada nivel SIL representa un nivel más alto de fiabilidad; cuanto más crítico sea el sistema, más alto será el SIL requerido.

Las primeras etapas del ciclo de vida de seguridad IEC 61508 definen el ámbito del sistema, evalúan las contingencias potenciales del sistema y estiman los riesgos que éstas presentan. A continuación tiene lugar la especificación de requerimientos de seguridad y la asignación de estos requerimientos de seguridad a los diferentes subsistemas. La actividad de

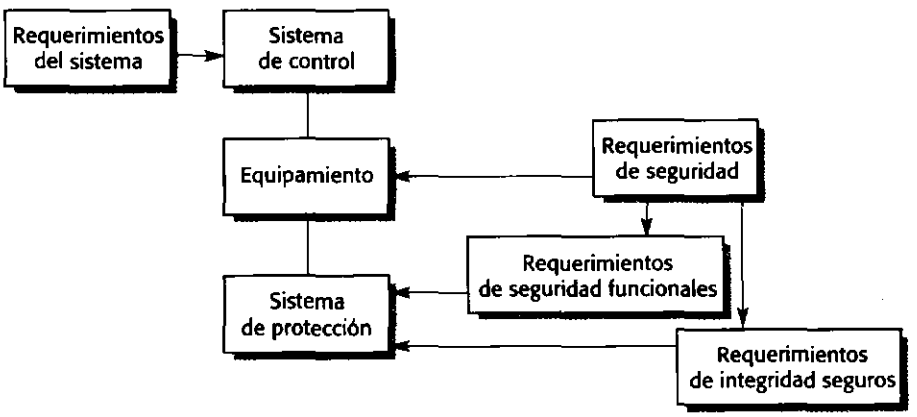


Figura 9.6 Requerimientos de seguridad de un sistema de control.

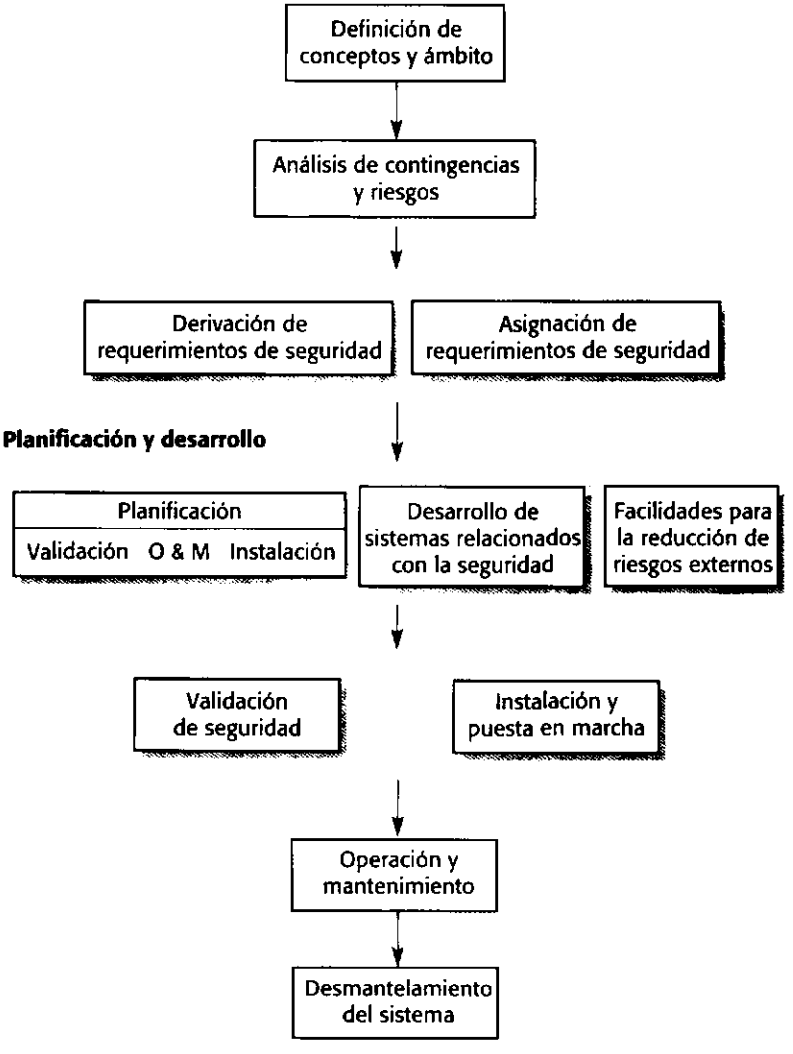


Figura 9.7 El ciclo de vida de seguridad IEC 61508.

desarrollo comprende la planificación y la implementación. El sistema de seguridad crítico se diseña e implementa, así como los sistemas externos relacionados que puedan proporcionar protección adicional. En paralelo a esto, se planifica la validación de la seguridad, instalación, operación y mantenimiento del sistema.

La gestión de la seguridad no termina al entregar el sistema. Después de la entrega, el sistema debe instalarse según lo planificado, de tal forma que el análisis de contingencias siga siendo válido. Después se lleva a cabo la validación de seguridad antes de que el sistema comience a utilizarse. La seguridad debe ser también gestionada durante el tiempo de funcionamiento del sistema y (particularmente) durante el mantenimiento del sistema. Muchos de los problemas relacionados con los sistemas de seguridad críticos tienen lugar debido a un proceso de mantenimiento pobre, por lo que es particularmente importante que el sistema se diseñe pensando en su mantenibilidad. Finalmente, también deberían tenerse en cuenta consideraciones de seguridad que podrían aplicarse durante el desmantelamiento del sistema (por ejemplo, desecho de material peligroso en tarjetas electrónicas).

9.3 Especificación de la protección

La especificación de los requerimientos de protección para los sistemas tiene algo en común con los requerimientos de seguridad. No es práctico especificarlos cuantitativamente, y los requerimientos de protección son a menudo requerimientos «no debería» que definen comportamientos inaceptables del sistema en lugar de funcionalidades requeridas del mismo. Sin embargo, existen diferencias importantes entre estos tipos de requerimientos:

1. Está bien desarrollada la noción de un ciclo de vida de seguridad que incluye todos los aspectos de la gestión de seguridad. El área de especificación y gestión de la protección todavía no está madura y no existe un equivalente aceptado de ciclo de vida de protección.
2. Si bien algunas amenazas de la protección son específicas del sistema, muchas son comunes a todos los tipos de sistemas. Todos los sistemas deben protegerse a sí mismos contra intrusiones, denegaciones de servicio, y otros. Por el contrario, las contingencias en sistemas de seguridad críticos son específicas del dominio.
3. Las técnicas y tecnologías de seguridad tales como encriptación y dispositivos de autenticación están bastante maduras. Sin embargo, el uso efectivo de esta tecnología a menudo requiere un alto nivel de sofisticación técnica. Dicha tecnología puede ser difícil de instalar, configurar y mantener actualizada. Como consecuencia, los gestores del sistema cometen errores ocasionando vulnerabilidades en el sistema.
4. El predominio de un suministrador de software en los mercados del mundo implica que un enorme número de sistemas se vea afectado por una violación de la protección en sus programas. La diversidad de la infraestructura informática es insuficiente y como consecuencia, es más vulnerable a amenazas externas. Generalmente, los sistemas de seguridad críticos están especializados y hechos a medida; por lo tanto, no puede darse la situación anterior.

La aproximación convencional (no informática) para el análisis de la protección se basa en los activos a proteger y su valor en una organización. Por lo tanto, un banco proporcionará alta seguridad en un área en la que se almacenen grandes cantidades de dinero en comparación con otras áreas públicas (por ejemplo) en donde las pérdidas potenciales están limitadas. La misma aproximación se puede utilizar para especificar la protección de sistemas informáticos. Un posible proceso de especificación de protección se muestra en la Figura 9.8.

Las etapas de este proceso son:

1. *Identificación y evaluación de activos.* Se identifican los activos (datos y programas) y su grado de protección requeridos. Dicha protección requerida depende del valor del activo, de forma que un fichero de contraseñas (por ejemplo) es normalmente más valioso que un conjunto de páginas web públicas, ya que un ataque con éxito sobre el fichero de contraseñas tendrá consecuencias serias en todo el sistema.
2. *Análisis de amenazas y valoración de riesgos.* Se identifican posibles amenazas de protección y se estiman los riesgos asociados con cada una de estas amenazas.
3. *Asignación de amenazas.* Las amenazas identificadas se relacionan con los activos de forma que para cada activo identificado exista una lista de amenazas.
4. *Análisis de la tecnología.* Se evalúan las tecnologías disponibles y su aplicabilidad contra las amenazas identificadas.

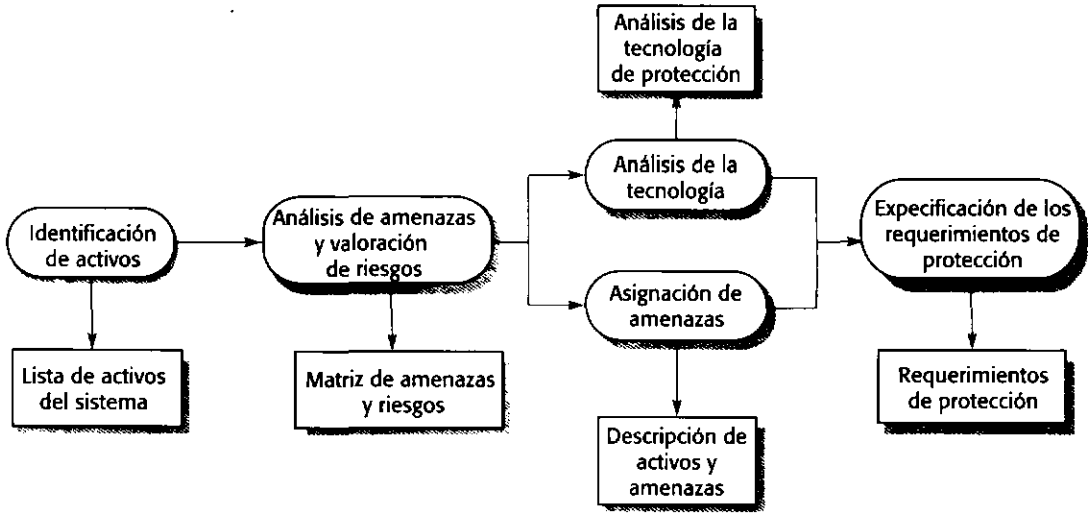


Figura 9.8 Especificación de la protección.

5. *Especificación de los requerimientos de protección.* Se especifican los requerimientos de protección. En donde sea apropiado, éstos identificarán explícitamente las tecnologías de protección que pueden utilizarse para proteger al sistema contra las amenazas identificadas.

La especificación de la protección y la gestión de la protección son esenciales para todos los sistemas críticos. Si un sistema no está protegido, entonces está sujeto a infecciones de virus y gusanos, corrupción y modificación no autorizada de datos, y ataques de denegación de servicio. Todo esto significa que no podemos estar seguros de que los esfuerzos realizados para asegurar la fiabilidad y seguridad sean efectivos.

Diferentes tipos de requerimientos de protección hacen referencia a distintas amenazas con las que se puede enfrentar el sistema. Firesmith (Firesmith, 2003) identifica 10 tipos de requerimientos de protección que pueden incluirse en un sistema:

1. *Los requerimientos de identificación* especifican si un sistema debería identificar a sus usuarios antes de interaccionar con él.
2. *Los requerimientos de autenticación* especifican cómo son identificados los usuarios.
3. *Los requerimientos de autorización* especifican los privilegios y permisos de acceso de los usuarios identificados.
4. *Los requerimientos de inmunidad* especifican cómo un sistema debería protegerse contra virus, gusanos, y amenazas similares.
5. *Los requerimientos de integridad* especifican cómo puede evitarse la corrupción de los datos.
6. *Los requerimientos de detección de intrusiones* especifican qué mecanismos deberían utilizarse para detectar ataques al sistema.
7. *Los requerimientos de no rechazo* especifican que una parte de la transacción no puede negar su pertenencia a dicha transacción.
8. *Los requerimientos de privacidad* especifican cómo se debería mantener la privacidad de los datos.
9. *Los requerimientos de auditoría de seguridad* especifican cómo puede auditarse y comprobarse el uso del sistema.

SEC1: Todos los usuarios del sistema deberían ser identificados utilizando su número de carnet de biblioteca y su contraseña personal.

SEC2: Los privilegios de los usuarios deberían ser asignados según el tipo de usuario (estudiante, personal administrativo, personal de biblioteca).

SEC3: Antes de ejecutar cualquier orden, LIBSYS debería comprobar que el usuario tenga suficientes privilegios para acceder y ejecutar dicha orden.

SEC4: Cuando un usuario pide un documento, la orden de petición debería ser registrada. Los datos de registro mantenidos deberían incluir la hora de la petición, la identificación del usuario y los artículos solicitados.

SEC5: Una vez al día se debería hacer una copia de seguridad de todos los datos del sistema y las copias de seguridad deberían guardarse en un área de almacenamiento segura.

SEC6: No debería permitirse a los usuarios tener simultáneamente más de un login para LIBSYS.

Figura 9.9 Requerimientos de protección para el sistema LIBSYS.

10. *Los requerimientos de seguridad de mantenimiento del sistema* especifican cómo una aplicación puede impedir los cambios que se hayan autorizado como consecuencia de una violación accidental de sus mecanismos de seguridad.

Por supuesto, no todos los sistemas necesitan todos estos requerimientos de protección. Los requerimientos específicos dependen del tipo de sistema, la forma de uso y los usuarios esperados. Como ejemplo, la Figura 9.9 muestra los requerimientos de protección que podrían incluirse en el sistema LIBSYS.

9.4 Especificación de la fiabilidad del software

La fiabilidad es un concepto complejo que siempre se debería considerar a nivel de sistema en lugar de a nivel de componente individual. Debido a que los componentes de un sistema son interdependientes, un fallo en un componente puede propagarse al resto del sistema y afectar al funcionamiento de otros componentes. En un sistema informático, se deben considerar tres dimensiones cuando se especifique la fiabilidad del sistema en su totalidad:

1. *Fiabilidad del hardware.* ¿Cuál es la probabilidad de que un componente hardware falle y cuánto tiempo se tarda en reparar dicho componente?
2. *Fiabilidad del software.* ¿Cuál es la probabilidad de que un componente software produzca una salida incorrecta? Los fallos del software son distintos de los fallos del hardware en tanto que el software no se desgasta: éste puede continuar funcionando correctamente después de que se haya producido un resultado incorrecto.
3. *Fiabilidad del operador.* ¿Cuál es la probabilidad de que el operador de un sistema cometa un error?

Todas estas dimensiones están estrechamente relacionadas. Los fallos del hardware pueden provocar la generación de señales espurias que estén fuera del rango de las entradas esperadas por el software. Después, el software puede comportarse de forma impredecible. El comportamiento no esperado del sistema puede confundir al operador y provocarle estrés. El operador puede entonces actuar de forma incorrecta y elegir entradas que son inadecuadas para la situación de fallo actual. Además estas entradas confunden al sistema y generan más errores. Por lo tanto, puede ocurrir una situación en la que un fallo recuperable de un subsis-

tema rápidamente pueda dar lugar a un problema serio que requiera una reinicialización completa del sistema.

La fiabilidad de los sistemas debería especificarse como un requerimiento no funcional que, idealmente, se exprese de forma cuantitativa utilizando una de las métricas expuestas en la sección siguiente. Para cumplir los requerimientos de fiabilidad no funcionales, puede ser necesario especificar requerimientos funcionales y de diseño adicionales que determinen cómo los fallos pueden ser evitados o tolerados. Ejemplos de estos requerimientos de fiabilidad son:

1. Se debería establecer un rango predefinido para todos los valores que introduce el operador, y el sistema debería comprobar que todas las entradas del operador están dentro de este rango predefinido.
2. Como parte del proceso de inicialización, el sistema debería comprobar los bloques defectuosos de todos los discos.
3. Se debería utilizar la programación de *n* versiones para implementar el sistema de control de frenado.
4. El sistema debería implementarse utilizando un subconjunto seguro de Ada y debería verificarse por medio de análisis estático.

No existen reglas sencillas que se puedan utilizar para derivar requerimientos fiables funcionales. En las organizaciones que desarrollan sistemas críticos, generalmente existe un conocimiento organizacional sobre posibles requerimientos de fiabilidad y cómo afectan éstos a la fiabilidad real de un sistema. Estas organizaciones pueden especializarse en tipos específicos de sistemas, como los sistemas de control ferroviario, de tal forma que los requerimientos de fiabilidad, una vez derivados, se reutilizan en un amplio rango de sistemas. Cuanto mayor es el nivel de integridad de la seguridad (comentado anteriormente), requerido en sistemas de seguridad críticos, es más probable que los requerimientos de fiabilidad sean más estrictos.

9.4.1 Métricas de fiabilidad

Las métricas de fiabilidad fueron diseñadas inicialmente para componentes hardware. Los fallos en los componentes hardware son inevitables debido a factores físicos tales como la abrasión mecánica y el calentamiento eléctrico. Los componentes tienen un periodo de vida limitado y esto se refleja en la métrica de fiabilidad de hardware que más se utiliza, tiempo medio entre fallos (MTTF). El MTTF es el tiempo medio durante el cual se espera que un componente funcione. Normalmente, un fallo de un componente hardware es permanente; por lo tanto también es significativo el tiempo medio de reparación (MTTR), que refleja el tiempo necesario para reparar o reemplazar el componente.

Sin embargo, estas métricas hardware no son directamente aplicables a la especificación de la fiabilidad del software debido a que los fallos de los componentes software son a menudo transitorios más que permanentes. Solamente se manifiestan con algunas entradas. Si los datos no están dañados, a menudo el sistema puede continuar funcionando después de que ocurra un fallo.

Las métricas que se han utilizado para especificar la fiabilidad y disponibilidad del software se muestran en la Figura 9.10. La elección de qué métrica debería utilizarse depende del tipo de sistema sobre el que se aplica y de los requerimientos del dominio de la aplicación. Algunos ejemplos de tipos de sistemas en donde se utilizan estas métricas son:

1. *Probabilidad de fallos en la petición*. Esta métrica es la más adecuada para sistemas en lo que los servicios se demandan a intervalos de tiempo impredecibles o relativa-

POFOD Probabilidad de fallos en la petición	**La probabilidad de que el sistema falle cuando se solicite una petición de servicio. Un POFOD de 0,001 significa que una de cada 1.000 peticiones de servicio conduce a un fallo.**
ROCOF Tasa de ocurrencia de fallos	**La frecuencia de ocurrencia con la que es probable que ocurra un comportamiento inesperado. Un ROCOF de 2/100 significa que es probable que ocurran dos fallos en cada 100 unidades de tiempo de funcionamiento. Esta métrica se conoce algunas veces como la intensidad de fallos.**
MTTF Tiempo medio entre fallos	**El tiempo medio entre fallos de sistema observados. Un MTTF de 500 significa que puede esperarse un fallo cada 500 unidades de tiempo.**
AVAIL Disponibilidad	**La probabilidad de que el sistema esté disponible para su utilización en un momento determinado. Una disponibilidad de 0,998 significa que el sistema es probable que esté disponible durante 998 de cada 1.000 unidades de tiempo.**

Figura 9.10 Métricas de fiabilidad.

mente largos y en los que existen serias consecuencias si el servicio pedido no se proporciona. Se podría utilizar para especificar sistemas de protección como la fiabilidad de un sistema de liberación de presión en una planta química o el sistema de apagado de emergencia en una planta de energía.

2. *Tasa de ocurrencia de fallos*. Esta métrica debería utilizarse en donde se hagan peticiones regulares de los servicios del sistema y en donde sea importante que estos servicios se proporcionen correctamente. Debería utilizarse en la especificación de un sistema de cajero automático que procesa transacciones de clientes o en un sistema de reservas de hoteles.
3. *Tiempo medio entre fallos*. Esta métrica debería utilizarse en sistemas con transacciones largas; esto es, en donde la gente utiliza el sistema durante largos periodos de tiempo. El MTTF debería ser mayor que la longitud media de cada transacción. Ejemplos de sistemas en los que podría utilizarse esta métrica son sistemas de procesamiento de texto y sistemas CAD.
4. *Disponibilidad*. Esta métrica debería utilizarse en sistemas que no se detienen y en los que los usuarios esperan que el sistema proporcione un servicio continuado. Ejemplos de tales sistemas son los sistemas de centralita telefónica y sistemas de señalización ferroviaria.

Hay tres tipos de mediciones que pueden realizarse cuando se valora la fiabilidad de un sistema:

1. El número de fallos del sistema dado un número de peticiones de servicios del sistema. Se usa para medir el POFOD.
2. El tiempo (o número de transacciones) transcurrido entre fallos del sistema. Se utiliza para medir el ROCOF y el MTTF.
3. El tiempo consumido en reparar o reiniciar el sistema cuando ocurre un fallo. Dado que el sistema debe estar disponible continuamente, éste se utiliza para medir el AVAIL.

Las unidades de tiempo que se pueden utilizar en estas métricas son fechas de calendario, tiempo de procesador o algunas unidades discretas, como por ejemplo el número de transacciones. En sistemas que emplean mucho tiempo esperando responder a una petición de servicio, como los sistemas de centralita telefónica, la unidad de tiempo que debería utilizarse

es el tiempo de procesador. Si se utilizan fechas de calendario, entonces esto también incluye el tiempo durante el que el sistema no está haciendo nada.

Las fechas de calendario son una unidad de tiempo adecuada para sistemas que están en continuo funcionamiento. Por ejemplo, los sistemas de monitorización tales como sistemas de alarma y otros tipos de sistemas de control de procesos pertenecen a esta categoría. Los sistemas que procesan transacciones tales como cajeros automáticos y sistemas de reservas de billetes de avión tienen una carga variable dependiendo de la hora del día. En estos casos, la unidad de «tiempo» utilizada debería ser el número de transacciones; por ejemplo, el ROCOF podría ser el número de transacciones sin éxito en *n* miles de transacciones.

9.4.2 Requerimientos de fiabilidad no funcionales

En muchos documentos de requerimientos del sistema, los requerimientos de fiabilidad no se especifican cuidadosamente. Las especificaciones de fiabilidad son subjetivas y no mensurables. Por ejemplo, frases como «El software debería ser fiable en condiciones normales de uso» carecen de significado. Frases cuasi cuantitativas como «El software deberá exhibir no más de *n* defectos/1.000 líneas» son igualmente inútiles. Es imposible medir el número de defectos/1.000 líneas de código puesto que no se puede decir cuándo se han descubierto todos los defectos. Además, la frase no significa nada en lo que se refiere al comportamiento dinámico del sistema. Son los fallos de funcionamiento del software y no los defectos del software los que afectan a la fiabilidad de un sistema.

Los tipos de fallos que pueden ocurrir son específicos del sistema y las consecuencias de un fallo del sistema dependen de la naturaleza de ese fallo. Cuando se redacte una especificación de fiabilidad, habría que identificar los diferentes tipos de fallos y pensar sobre si éstos deberían ser tratados de forma diferente en la especificación. En la Figura 9.11 se muestran ejemplos de diferentes tipos de fallos. Obviamente pueden ocurrir combinaciones de éstos, como por ejemplo un fallo que sea transitorio, recuperable y corruptivo.

La mayoría de los sistemas grandes están compuestos por varios subsistemas con diferentes requerimientos de fiabilidad. Puesto que el software que tiene una fiabilidad alta es caro, debería realizarse una valoración de los requerimientos de fiabilidad para cada subsistema por separado en lugar de imponer el mismo requerimiento de fiabilidad para todos los subsistemas. Esto evita imponer altos requerimientos de fiabilidad en aquellos subsistemas en los que no es necesario.

Los pasos que se requieren para establecer una especificación de la fiabilidad son:

1. Para cada subsistema, identificar los tipos de fallos de funcionamiento del sistema que pueden ocurrir y analizar las consecuencias de dichos fallos.

Transitorio	**Ocurre solamente con ciertas entradas.**
Permanente	**Ocurre con todas las entradas.**
Recuperable	**El sistema puede recuperarse sin la intervención del operador.**
Irrecuperable	**Es necesaria la intervención del operador para recuperarse del fallo.**
No corruptivo	**El fallo no corrompe el estado del sistema o los datos.**
Corruptivo	**El fallo corrompe el estado del sistema o los datos.**

Figura 9.11 Clasificación de fallos de funcionamiento.

2. A partir del análisis de fallos del sistema, dividir los fallos en clases. Un punto de partida razonable es utilizar los tipos de fallos mostrados en la Figura 9.11.
3. Para cada clase de fallo identificada, definir el requerimiento de fiabilidad utilizando una métrica de fiabilidad adecuada. No es necesario utilizar la misma métrica para diferentes clases de fallos. Si un fallo requiere alguna intervención para poder recuperar el sistema, la métrica más apropiada podría ser la probabilidad de fallos en la petición. Cuando es posible la recuperación automática y el efecto del fallo es la causa de una molestia para el usuario, ROCOF podría ser la más adecuada.
4. Donde sea apropiado, identificar los requerimientos de fiabilidad funcionales que definen la funcionalidad del sistema para reducir la probabilidad de fallos críticos.

Como ejemplo, considere los requerimientos de fiabilidad para un cajero automático (ATM). Suponga que cada máquina en la red se utiliza alrededor de 300 veces al día. El tiempo de vida del hardware del sistema es 5 años y el software se actualiza normalmente cada año. Por lo tanto, durante el tiempo de vida de una versión del software comercializada, cada máquina gestionará alrededor de 100.000 transacciones. Un banco tiene 1.000 máquinas en su red. Esto significa que hay 300.000 transacciones sobre la base de datos central al día (es decir, 100 millones por año).

La Figura 9.12 muestra posibles clases de fallos y posibles especificaciones de fiabilidad para diferentes tipos de fallos del sistema. Los requerimientos de fiabilidad establecen que es aceptable que una caída permanente en una máquina ocurra una vez cada tres años. Esto significa que, en promedio, una máquina de la red bancaria podría verse afectada cada día. Por el contrario, los defectos que conllevan que una transacción sea cancelada ocurren con relativa frecuencia. El único efecto de dichos fallos solamente es una inconveniencia menor al usuario.

En condiciones ideales, los defectos que corrompen la base de datos nunca deberían ocurrir durante el tiempo de vida del software. Por lo tanto, el requerimiento de fiabilidad que podría utilizarse para esto es que la probabilidad de que ocurra un fallo corruptivo cuando se realiza una petición es menor que 1 en 200 millones de transacciones. Esto es, durante el tiempo de vida de una versión del software para un ATM, nunca debería ocurrir un error que provocara la corrupción de la base de datos.

Sin embargo, un requerimiento de fiabilidad como éste no puede realmente probarse. Consideremos, por ejemplo, que cada transacción consume un segundo de tiempo de máquina y que puede construirse un simulador para la red de ATMs. La simulación de las transacciones que tienen lugar en un solo día en toda la red tardará un tiempo de 300.000 segundos. Esto es aproximadamente 3,5 días. Claramente este periodo puede reducirse disminuyendo el tiem-

Permanente, no corruptivo	**El sistema deja de funcionar con cualquier tarjeta que se introduzca. El software debe reinicializarse para corregir el fallo.**	**ROCOF 1 ocurrencia/1.000 días**
Transitorio, no corruptivo	**La banda magnética de datos de una tarjeta no dañada no puede leerse.**	**ROCOF 1 en 1.000 transacciones**
Transitorio, corruptivo	**Un patrón de transacciones en la red provoca que la base de datos se corrompa.**	**¡No cuantificable! No debería ocurrir nunca durante el tiempo de vida del sistema.**

Figura 9.12 Especificación de fiabilidad para un ATM.

po de transacción y utilizando varios simuladores. Pero, aun así, es muy difícil probar el sistema para validar la especificación de fiabilidad.

Es imposible validar requerimientos cualitativos que requieren un alto nivel de fiabilidad. Por ejemplo, pensemos en un sistema desarrollado para utilizarse en una aplicación de seguridad crítico. Dicho sistema no debería fallar nunca durante el tiempo total de vida del sistema. Suponga que deben instalarse 1.000 copias de dicho sistema, y que éste se «ejecuta» 1.000 veces por segundo. El tiempo de vida estimado del sistema es de 10 años. El número total estimado de ejecuciones del sistema es aproximadamente $3 * 10^{14}$. No merece la pena especificar que la tasa de ocurrencia de fallos debería ser de $1/10^{15}$ ejecuciones (esto permite algún factor de seguridad) puesto que usted no puede probar el sistema durante todo este tiempo para validar este nivel de fiabilidad.

Como otro ejemplo adicional, considere los requerimientos de fiabilidad para el sistema de suministro de insulina. El sistema suministra insulina varias veces al día y monitoriza el nivel de glucosa en la sangre varias veces por hora. Debido a que el uso del sistema es intermitente y las consecuencias de un fallo son serias, la métrica de fiabilidad más adecuada es POFOD (probabilidad de fallo en la petición).

Un fallo en el suministro de insulina no tiene implicaciones de seguridad inmediatas, por lo que los factores comerciales más que los de seguridad determinan el nivel de fiabilidad requerida. Los costes del servicio son altos debido a que los usuarios necesitan un servicio de reparación y sustitución rápido. Es obligación del fabricante limitar el número de fallos permanentes que requieren reparación.

De nuevo, se pueden identificar dos tipos de fallos:

1. *Fallos transitorios* que se pueden reparar mediante acciones del usuario, como reinicializar o recalibrar la máquina. Para estos tipos de fallos, puede ser aceptable un valor relativamente bajo de POFOD (por ejemplo, 0,002). Esto significa que un fallo puede ocurrir cada 500 peticiones realizadas a la máquina. Esto es aproximadamente una vez cada 3,5 días.
2. *Fallos permanentes* que requieren que la máquina sea reparada por el fabricante. La probabilidad de este tipo de fallos debería ser mucho menor. Aproximadamente una vez al año es lo mínimo, por lo que POFOD no debería ser mayor de 0,00002.

PUNTOS CLAVE

- **El análisis de riesgos es una actividad clave en el proceso de especificación de sistemas críticos. Dicho análisis implica la identificación de riesgos que pueden provocar accidentes o incidentes. Los requerimientos del sistema se establecen entonces para asegurar que estos riesgos no ocurran o, si ocurren, no provoquen un incidente.**
- **El análisis de riesgos es el proceso de valorar la probabilidad de que un riesgo provoque un accidente. El análisis de riesgos identifica riesgos críticos en el sistema que deberían evitarse y los clasifica de acuerdo con su gravedad.**
- **Para especificar los requerimientos de seguridad, se deberían identificar los activos que tienen que protegerse y definir cómo deberían utilizarse las técnicas y tecnologías de protección para proteger estos activos.**

- **Los requerimientos de fiabilidad deberían definirse de forma cuantitativa en la especificación de los requerimientos del sistema.**
- **Existen varias métricas de fiabilidad, tal como la probabilidad de fallos en la petición (POFOD), tasa de ocurrencia de fallos, tiempo medio entre fallos (MTTF) y disponibilidad. La métrica más apropiada para un sistema específico depende del tipo de sistema y del dominio de la aplicación. Pueden utilizarse métricas diferentes para subsistemas diferentes.**
- **Las especificaciones de fiabilidad no funcionales pueden conducir a requerimientos funcionales del sistema que definen las características del mismo y cuya función es reducir el número de fallos del sistema y, por lo tanto, incrementar la fiabilidad.**
- **El coste de desarrollar y validar una especificación de fiabilidad del sistema puede ser muy alto. Las organizaciones deben ser realistas sobre si estos costes merecen la pena. Éstos están claramente justificados en sistemas en donde es crítico un funcionamiento fiable, como sistemas de centralita telefónica o en sistemas en donde los fallos pueden provocar grandes pérdidas económicas. Probablemente no esté justificado por muchos tipos de sistemas científicos o de negocio. Éstos tienen requerimientos de fiabilidad modestos, ya que los costes de fallo son simplemente retrasos en el procesamiento, y es sencillo y relativamente barato recuperarse de dichos fallos.**

LECTURAS ADICIONALES

«Security use cases». Un buen artículo, disponible en la web, que se centra en cómo los casos de uso se pueden utilizar en la especificación de la protección. El autor también tiene varios buenos artículos sobre especificación de seguridad a los que hace referencia en este artículo. [D. G. Firesmith, *Journal of Object Technology*, 2 (3), mayo-junio de 2003.]

«Requeriments Definition for survivable network systems». Trata los problemas de definición de requerimientos para sistemas en los que es importante la supervivencia, relacionada ésta con la disponibilidad y la protección. (R. C. Linger *et al.*, *Proc. ICRE'98*, IEEE Press, 1998.)

Requirements Engineering: A Good Practice Guide. Este libro incluye una sección sobre la especificación de sistemas críticos y un estudio del uso de métodos formales en la especificación de sistemas críticos. (I. Sommerville y P. Sawyer, 1997, John Wiley and Sons.)

Safeware: System Safety and Computers. Éste es un estudio completo de todos los aspectos de los sistemas de seguridad críticos. Hace especial hincapié en su descripción del análisis de contingencias y la derivación de requerimientos a partir de dicho análisis. (N. Leveson, 1995, Addison-Wesley.)

EJERCICIOS

9.1 Explique por qué los límites del triángulo de riesgos mostrado en la Figura 9.2 es susceptible de cambio con el tiempo y con las actitudes sociales cambiantes.

9.2 En el sistema de suministro de insulina, el usuario tiene que cambiar la aguja y suministrar insulina en intervalos regulares y también puede cambiar la dosis máxima para un día o bien la dosis máxima diaria que se puede suministrar. Sugiera tres errores de usuario que podrían ocurrir y proponga requerimientos de seguridad que podrían evitar que estos errores produjeran un accidente.

9.3 Un sistema software de seguridad crítico para el tratamiento de pacientes con cáncer tiene dos componentes principales:

- La máquina de terapia de radiación que suministra dosis controladas de radiación a los puntos del tumor. Esta máquina se controla por medio de un sistema software embebido.
- Una base de datos de tratamientos que incluye los detalles del tratamiento dado a cada paciente. Los requerimientos del tratamiento se introducen en esta base de datos y automáticamente se descargan en la máquina para la terapia de radiación.

Identifique tres contingencias que puedan surgir en este sistema. Para cada contingencia sugiera un requerimiento defensivo que reduzca la probabilidad de que estas contingencias provoquen un accidente. Explique por qué la defensa sugerida por usted es probable que reduzca el riesgo asociado a la contingencia.

9.4 Describa tres diferencias importantes entre los procesos de especificación de seguridad y especificación de protección.

9.5 Sugiera cómo puede modificarse un análisis de árbol de defectos para ser usado en la especificación de la protección. Las amenazas en un sistema de protección crítico son análogas a las contingencias en un sistema de seguridad crítico.

9.6 ¿Cuál es la diferencia fundamental entre los fallos de funcionamiento del hardware y del software? Dada esta diferencia, explique por qué las métricas de fiabilidad del hardware son a menudo inadecuadas para medir la fiabilidad del software.

9.7 Explique por qué es prácticamente imposible validar las especificaciones de fiabilidad cuando éstas se expresan en términos de un número muy pequeño de fallos sobre el tiempo total de vida de un sistema.

9.8 Sugiera métricas de fiabilidad apropiadas para las siguientes clases de sistemas software. Dé razones para su elección de dichas métricas. Prediga el uso de estos sistemas y sugiera valores apropiados para las métricas de fiabilidad:

- Un sistema que monitoriza pacientes en una unidad de cuidados intensivos de un hospital.
- Un procesador de texto.
- Un sistema de control de máquinas expendedoras automáticas.
- Un sistema de control de frenado en un coche.
- Un sistema para controlar una unidad de refrigeración.
- Un generador de informes administrativos.

9.9 Usted es responsable de escribir la especificación para un sistema software que controla una red de terminales EPOS (punto de venta electrónico) en un almacén. El sistema acepta información de código de barras desde un terminal, consulta una base de datos de productos y devuelve el nombre del artículo y su precio al terminal para su visualización. El sistema debe estar disponible continuamente durante las horas en las que está abierto el almacén.

Elija las métricas apropiadas para especificar la fiabilidad de dicho sistema justificando su elección, y escriba una especificación de fiabilidad plausible que tenga en cuenta el hecho de que algunos defectos son más serios que otros.

Sugiera cuatro requerimientos funcionales que podrían generarse para este sistema de almacén para mejorar la fiabilidad del sistema.

9.10 Un sistema de protección ferroviario frena automáticamente un tren si el límite de velocidad se excede en un tramo de vía o si el tren entra en un tramo de vía que está señalizado con una luz roja (es decir, no se debería haber entrado en ese tramo). Elija una métrica de fiabilidad, justificando su respuesta, que podría usarse para especificar la fiabilidad requerida para dicho sistema.

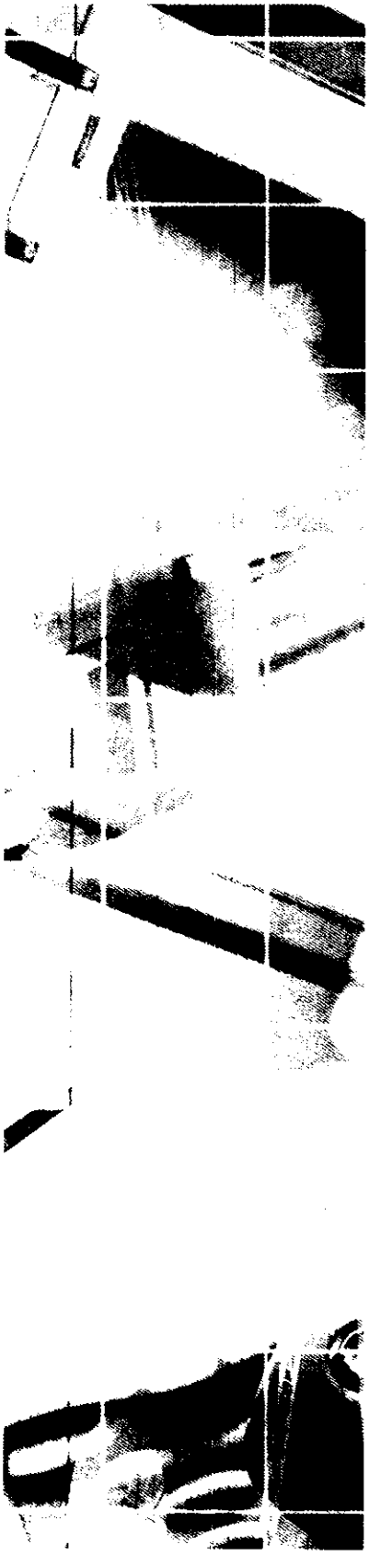
Hay dos requerimientos esenciales de seguridad para dicho sistema:

- El tren no debería entrar en un tramo de pista señalizado con una luz roja.
- El tren no debería exceder el límite de velocidad especificado para un tramo de vía.

Suponiendo que el estado de la señal luminosa y el límite de velocidad para el tramo de vía se transmite desde el software del cuadro de mandos al tren antes de que éste entre en un tramo de pista, proponga cinco posibles requerimientos funcionales del sistema para el software del cuadro de mandos que puedan derivarse de los requerimientos de seguridad del sistema.

9.11 Los ingenieros software que trabajan en la especificación y desarrollo de sistemas relacionados con la seguridad ¿deberían estar certificados profesionalmente de alguna forma? Justifique su respuesta.

9.12 Como experto en protección de computadoras, usted ha sido requerido por una organización que realiza una campaña por los derechos de las víctimas de torturas y le han pedido que ayude a la organización a conseguir accesos no autorizados a los sistemas informáticos de una compañía británica. Esto les ayudaría a confirmar o desmentir que esta compañía está vendiendo equipamiento que se usa directamente en la tortura de prisioneros políticos. Comente los dilemas éticos que esta solicitud provoca y cómo podría usted reaccionar ante esta petición.



10 Especificación formal

Objetivos

El objetivo de este capítulo es introducir las técnicas de especificación formal que se pueden usar para añadir detalle a una especificación de requerimientos de un sistema. Cuando haya leído este capítulo:

- comprenderá por qué las técnicas de especificación formal ayudan a descubrir problemas en los requerimientos del sistema;
- comprenderá el uso de técnicas algebraicas de especificación formal para definir las especificaciones de interfaz;
- comprenderá cómo las técnicas formales basadas en modelos formales se usan para especificar el comportamiento.

Contenidos

En las disciplinas de ingeniería «tradicionales», tales como la ingeniería civil y la eléctrica, el progreso normalmente ha conducido al desarrollo de mejores técnicas matemáticas. La industria de ingeniería no ha tenido dificultad en aceptar la necesidad del análisis matemático y en incorporar éste en sus procesos. El análisis matemático es una parte rutinaria del proceso de desarrollo y validación del diseño de un producto.

Sin embargo, la ingeniería del software no ha seguido el mismo camino. A pesar de que hasta ahora han sido más de 30 años de investigación en la utilización de técnicas matemáticas en el proceso del software, estas técnicas han tenido un impacto limitado. Los denominados métodos formales de desarrollo del software no son muy utilizados en el desarrollo de software industrial. La mayoría de las compañías de desarrollo de software no consideran rentable aplicarlos a sus procesos del software.

La denominación *métodos formales* se usa para referirse a cualquier actividad relacionada con representaciones matemáticas del software, incluyendo la especificación formal de sistemas, análisis y demostración de la especificación, el desarrollo transformacional y la verificación de programas. Todas estas actividades dependen de una especificación formal del software. Una especificación formal del software es una especificación expresada en un lenguaje cuyo vocabulario, sintaxis y semántica están formalmente definidos. Esta necesidad de una definición formal significa que los lenguajes de especificación deben basarse en conceptos matemáticos cuyas propiedades se comprendan bien. La rama de las matemáticas usada es la de matemática discreta, y los conceptos matemáticos provienen de la teoría de conjuntos, la lógica y el álgebra.

En la década de los 80, muchos investigadores de ingeniería del software propusieron que el uso de métodos formales de desarrollo era la mejor forma de mejorar la calidad del software. Argumentaban que el rigor y el análisis detallado, que son una parte esencial de los métodos formales, podrían dar lugar a programas con menos errores y más adecuados a las necesidades de los usuarios. Predijeron que, en el siglo XXI, una gran proporción del software estaría desarrollado usando métodos formales.

Claramente, esta predicción no se ha hecho realidad. Existen cuatro razones principales para esto:

1. *Una ingeniería del software exitosa.* El uso de otros métodos de ingeniería del software como los métodos estructurados, gestión de configuraciones y ocultación de la información en el diseño del software y procesos de desarrollo ha conducido a mejoras en la calidad del software. La gente que sugirió que la única forma de mejorar la calidad del software era usando métodos formales estaba claramente equivocada.
2. *Cambios en el mercado.* En la década de los 80, la calidad del software fue vista como un problema clave de la ingeniería del software. Sin embargo, desde entonces, la cuestión crítica para muchas clases de desarrollo del software no es la calidad, sino la oportunidad de mercado. El software debe desarrollarse rápidamente, y los clientes están dispuestos a aceptar software con algunos defectos si se les entrega rápidamente. Las técnicas para el desarrollo rápido del software no funcionan de forma efectiva con las especificaciones formales. Por supuesto, la calidad todavía es un factor importante, pero debe lograrse en el contexto de entrega rápida.
3. *Ámbito limitado de los métodos formales.* Los métodos formales no son muy apropiados para la especificación de interfaces de usuario e interacciones del usuario. El componente de interfaz de usuario se ha convertido en una parte cada vez mayor de la mayoría de los sistemas, de manera que realmente sólo pueden usarse métodos formales cuando se desarrollan las otras partes del sistema.

4. *Escalabilidad limitada de los métodos formales.* Los métodos formales todavía no son muy escalables. La mayoría de los proyectos con éxito que han usado estas técnicas han estado relacionados con núcleos de sistemas críticos relativamente pequeños. A medida que los sistemas incrementan su tamaño, el tiempo y esfuerzo requerido para desarrollar una especificación formal crece de forma desproporcionada.

Estos factores implican que la mayoría de las empresas que desarrollan software no estén dispuestas a arriesgarse a usar métodos formales en sus procesos de desarrollo. Sin embargo, la especificación formal es una forma excelente de descubrir errores en la especificación y presentar la especificación del sistema de forma no ambigua. Las organizaciones que han hecho una inversión para el uso de métodos formales presentan menos errores en el software entregado sin un incremento en los costes de desarrollo. Parece que los métodos formales pueden ser rentables si su uso se limita a partes del núcleo del sistema y si las compañías están dispuestas a realizar una alta inversión inicial en esta tecnología.

El uso de métodos formales está creciendo en el área del desarrollo de sistemas críticos, en donde las propiedades emergentes del sistema tales como seguridad, fiabilidad y protección son muy importantes. El alto coste de los fallos de funcionamiento en estos sistemas implica que las compañías están dispuestas a aceptar los costes elevados iniciales de los métodos formales para asegurar que su software es tan confiable como sea posible. Tal y como se comenta en el Capítulo 24, los sistemas críticos tienen unos costes de validación muy altos, y los costes de fallos de funcionamiento del sistema son importantes y crecientes. Los métodos formales pueden reducir estos costes.

Los sistemas críticos en los que los métodos formales se han aplicado con éxito incluyen un sistema de información de control de tráfico aéreo (Hall, 1996), sistemas de señalización de redes ferroviarias (Dehbonei y Mejia, 1995), sistemas de naves espaciales (Easterbrook *et al.*, 1998) y sistemas de control médico (Jacky *et al.*, 1997; Jacky, 1995). Los métodos formales han sido utilizados también para la especificación de herramientas software (Neil *et al.*, 1998), la especificación de parte de los sistemas CICS de IBM (Wordsworth, 1991) y un kernel de un sistema de tiempo real (Spivey, 1990). El método de sala limpia (*Cleanroom method*) de desarrollo de software (Prowell *et al.*, 1999) utiliza argumentos formales que son codificados de acuerdo con su especificación. Debido a que el razonamiento sobre la protección de un sistema también es posible si se desarrolla una especificación formal, es probable que los sistemas protegidos constituyan un área importante para el uso de los métodos formales (Hall y Chapman, 2002).

10.1 Especificación formal en el proceso del software

El desarrollo de sistemas críticos normalmente implica un proceso de software que utiliza un plan basado en el modelo de ciclo de desarrollo en cascada explicado en el Capítulo 4. Tanto los requerimientos del sistema como el diseño se expresan con detalle y son analizados cuidadosamente antes de que comience la implementación. Si se desarrolla una especificación formal del software, ésta normalmente tiene lugar después de que se hayan especificado los requerimientos del sistema, pero antes del diseño detallado de dicho sistema. Aquí hay un estrecho bucle de realimentación entre la especificación detallada de los requerimientos y la especificación formal. Tal y como se indica más adelante, uno de los beneficios de la especificación formal es la capacidad para descubrir problemas y ambigüedades en los requerimientos del sistema.

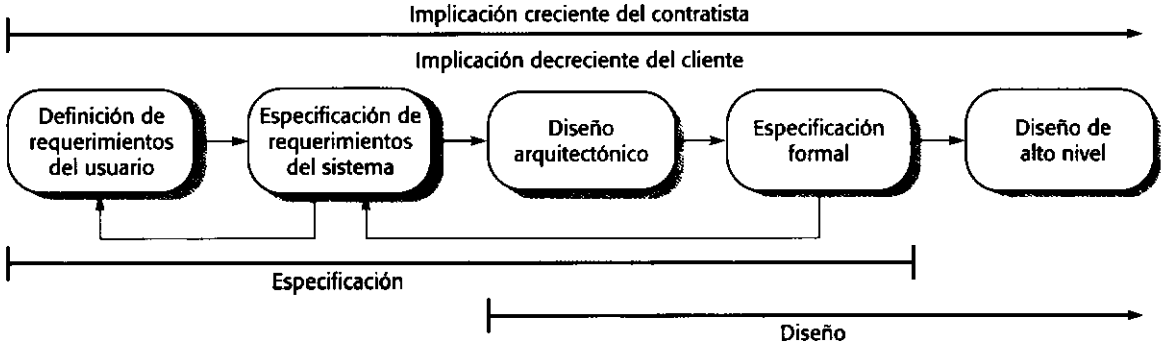


Figura 10.1 Especificación y diseño.

La implicación del cliente disminuye y la implicación del contratante del software se incrementa a medida que se añade más detalle a la especificación del sistema. En etapas tempranas del proceso, la especificación debería ser «orientada al cliente». Debería redactarse la especificación para que el cliente la pueda comprender, y debería hacerse el menor número de asunciones posibles sobre el diseño del software. Sin embargo, la etapa final del proceso, que es la construcción de una especificación completa, consistente y precisa, está orientada principalmente al contratante del software. Éste especifica los detalles de la implementación del sistema. Puede utilizarse un lenguaje formal en esta etapa para evitar la ambigüedad en la especificación del software.

La Figura 10.1 muestra las etapas de la especificación del software y su interfaz con el proceso de diseño. Estas etapas de especificación no son independientes ni es preciso que se desarrollen en la secuencia indicada. La Figura 10.2 muestra las actividades de especificación y diseño que pueden llevarse a cabo de forma paralela. Existe una relación bidireccional entre cada etapa del proceso. La información circula desde el proceso de especificación al de diseño y viceversa.

A medida que se desarrolla la especificación con detalle, se incrementa el conocimiento sobre dicha especificación. La creación de una especificación formal obliga a realizar un análisis detallado del sistema que normalmente revela errores e incongruencias en la especificación informal de requerimientos. Esta detección de errores es probablemente el argumento más potente para desarrollar una especificación formal (Hall, 1990). La especificación formal ayuda a descubrir problemas de los requerimientos que pueden ser muy caros de corregir más tarde.

Dependiendo del proceso usado, los problemas de la especificación descubiertos durante el análisis formal podrían conducir a cambios en la especificación de requerimientos si ésta no ha sido acordada. Si la especificación de requerimientos ha sido acordada y se incluye en

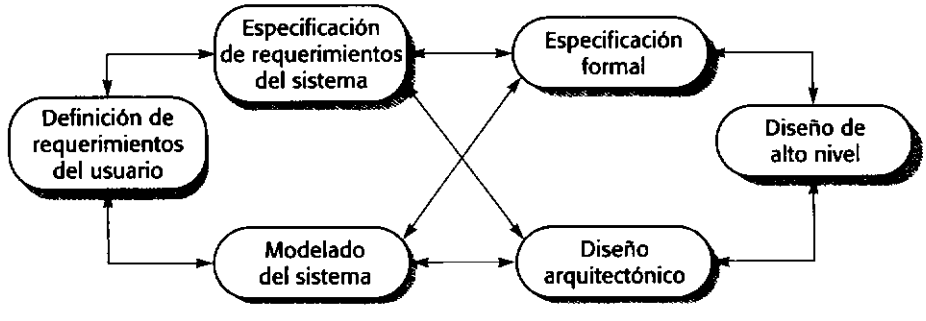


Figura 10.2 Especificación formal en el proceso del software.

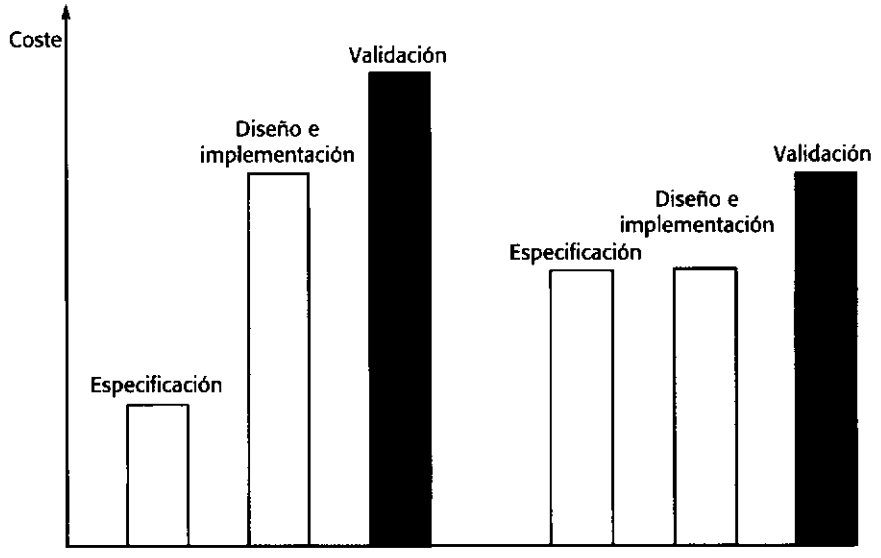


Figura 10.3
Costes de desarrollo del software con especificación formal.

el contrato del desarrollo del sistema, habría que plantear al cliente los problemas que ha encontrado. Este último es entonces quien debería decidir cómo tendrían que resolverse antes de empezar con el proceso de diseño del sistema.

El desarrollo y análisis de una especificación formal desplaza las cargas de los costes de desarrollo hacia las primeras etapas del mismo. La Figura 10.3 muestra cómo los costes del proceso del software es probable que se vean afectados por el uso de una especificación formal. Cuando se usa un proceso convencional, los costes de validación constituyen alrededor del 50% de los costes de desarrollo, y los costes de implementación y diseño constituyen alrededor de dos veces los costes de especificación. Con la especificación formal, los costes de especificación e implementación son comparables, y los costes de validación del sistema se reducen de forma significativa. Así como el desarrollo de una especificación formal descubre problemas en los requerimientos, también se evita el volver a realizar el trabajo para corregir dichos problemas una vez diseñado el sistema.

Se han utilizado dos aproximaciones fundamentales para redactar especificaciones detalladas para sistemas de software industriales. Éstas son:

1. *Una aproximación algebraica*, en la que el sistema se describe en función de las operaciones y sus relaciones.
2. *Una aproximación basada en modelos*, en la que se construye un modelo del sistema utilizando construcciones matemáticas como conjuntos y sucesiones, y las operaciones del sistema se definen indicando cómo éstas modifican el estado del sistema.

Se han desarrollado diferentes lenguajes dentro de estas dos aproximaciones para especificar sistemas concurrentes y secuenciales. La Figura 10.4 muestra ejemplos de lenguajes en

Algebraico	**Larch (Guttag *et al.*, 1993)** **OBJ (Futatsugi *et al.*, 1985)**	**Lotos (Bolognesi y Brinksma, 1987)**
Basado en modelos	**Z (Spivey, 1992)** **VDM (Jones, 1980)** **B (Wordsworth, 1996)**	**CSP (Hoare, 1985)** **Petri Nets (Peterson, 1981)**

Figura 10.4
Lenguajes de especificación formal.

cada una de estos enfoques. Se puede ver en esta tabla que la mayoría de estos lenguajes fueron desarrollados en la década de los 80. Lleva varios años refinar un lenguaje de especificación formal, por lo que la mayor parte de la investigación en especificación formal se basa actualmente en estos lenguajes más que interesarse en inventar notaciones nuevas.

El objetivo de este capítulo es introducir ambas aproximaciones, tanto la algebraica como la basada en modelos. Los ejemplos mostrados deberían dar una idea de cómo la especificación formal produce una especificación precisa y detallada, pero no se pretende describir los detalles del lenguaje de especificación, ni las técnicas de especificación o los métodos de verificación de programas. El lector puede descargar una descripción más detallada de ambas técnicas desde el sitio web del libro.

10.2 Especificación de interfaces de subsistemas

Los grandes sistemas se descomponen normalmente en subsistemas que se desarrollan de forma independiente. Los subsistemas usan otros subsistemas; por lo tanto, una parte esencial del proceso de especificación es la definición de interfaces de subsistemas. Una vez que los interfaces se han acordado y definido, los subsistemas pueden entonces diseñarse e implementarse de forma independiente.

Los interfaces de subsistemas se definen a menudo como un conjunto de objetos o componentes (Figura 10.5). Éstos describen los datos y operaciones a los que puede acceder a través de la interfaz del subsistema. Por lo tanto, se puede definir una especificación de la interfaz de un subsistema combinando las especificaciones de los objetos que componen la interfaz.

Es importante realizar especificaciones precisas de subsistemas debido a que los desarrolladores de los subsistemas deben escribir código que utiliza los servicios de otros subsistemas antes de que éstos hayan sido implementados. La especificación de la interfaz proporciona información a los desarrolladores de subsistemas para que éstos conozcan qué servicios estarán disponibles en otros subsistemas y cómo se puede acceder a ellos. Las especificaciones de interfaces de subsistemas claras y no ambiguas reducen la posibilidad de malentendidos entre un subsistema que proporciona algún servicio y el subsistema que usa dicho servicio.

La aproximación algebraica fue diseñada originalmente para la definición de interfaces de tipos abstractos de datos. En un tipo abstracto de datos, el tipo se define especificando el tipo de operaciones en lugar del tipo de representación. Por lo tanto, esto es similar a una clase de objeto. El método algebraico de especificación formal define el tipo abstracto de datos en función de las relaciones entre los tipos de operaciones.

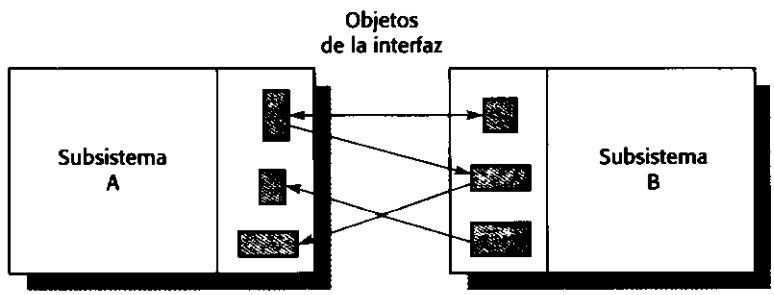


Figura 10.5 Objetos de la interfaz del subsistema.

Guttag (Guttag, 1977) fue el primero en proponer esta aproximación para la especificación de tipos abstractos de datos. Cohen y otros (Cohen *et al.*, 1986) muestran cómo la técnica puede ampliarse para completar la especificación del sistema usando un ejemplo de un sistema de recuperación de documentos. Liskov y Guttag (Liskov y Guttag, 1986) también tratan la especificación algebraica de los tipos abstractos de datos.

La estructura de la especificación de un objeto se muestra en la Figura 10.6. El cuerpo de la especificación tiene cuatro componentes:

1. *Una introducción* que declara la clase (el nombre del tipo) de la entidad que se está especificando. Una clase es el nombre de un conjunto de objetos con características comunes. Es similar a un tipo en un lenguaje de programación. La introducción también puede incluir una declaración «imports», en la que se declaran los nombres de la especificación que definen otras clases. Importar una especificación hace que estas clases estén disponibles para ser usadas.
2. *Una parte de descripción*, en donde las operaciones se describen informalmente. Esto hace que la especificación formal sea más fácil de entender. La especificación formal complementa esta descripción proporcionando una sintaxis y semántica no ambiguas para las operaciones del tipo.
3. *La parte de signatura* define la sintaxis de la interfaz de la clase del objeto o tipo abstracto de datos. La signatura describe los nombres de las operaciones que se definen, el número y clase de sus parámetros, y las clases de los resultados de las operaciones.
4. *La parte de axiomas* define la semántica de las operaciones mediante la definición de un conjunto de *axiomas* que caracterizan el comportamiento del tipo abstracto de datos. Estos axiomas relacionan las operaciones utilizadas para construir entidades de la clase definida con las operaciones empleadas para inspeccionar sus valores.

El proceso del desarrollo de una especificación formal de la interfaz de un subsistema comprende las siguientes actividades:

1. *Estructura de la especificación*. Se organiza la especificación informal de la interfaz en un conjunto de tipos abstractos de datos o clases de objetos. Se deberían definir informalmente las operaciones asociadas con cada clase.
2. *Nombrado de la especificación*. Se establece un nombre para la especificación de cada tipo abstracto de datos, decide si éstos requieren parámetros y determina los nombres para las clases identificadas.
3. *Selección de las operaciones*. Se elige un conjunto de operaciones para cada especificación basada en la funcionalidad identificada de la interfaz. Deberían incluirse operaciones para crear instancias de la clase, para modificar el valor de las instancias y para inspeccionar los valores de las instancias. Deben añadirse funciones a las inicialmente identificadas en la definición informal de la interfaz.

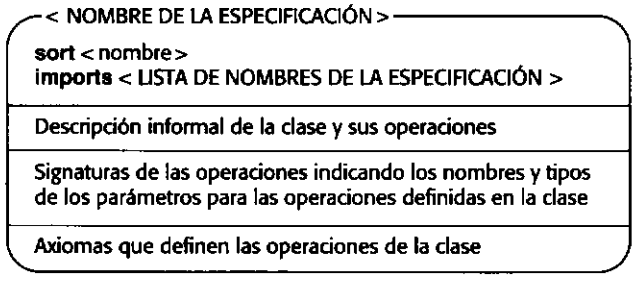


Figura 10.6
La estructura de una especificación algebraica.

4. *Especificación informal de las operaciones.* Se redacta una especificación informal de cada operación. Debería describirse cómo las operaciones afectan a la clase definida.
5. *Definición de la sintaxis.* Se define la sintaxis de las operaciones y sus parámetros. Ésta es la parte de la signatura de la especificación formal. Si fuera necesario, debería actualizarse la especificación informal en esta etapa.
6. *Definición de axiomas.* Se define la semántica de las operaciones describiendo qué condiciones son siempre verdaderas para diferentes combinaciones de operaciones.

Para explicar la técnica de especificación algebraica, se utiliza un ejemplo de una estructura de datos sencilla (una lista enlazada), tal y como se muestra en la Figura 10.7. Las listas enlazadas son estructuras de datos ordenados en donde cada elemento incluye un enlace al siguiente elemento de la estructura. Se ha usado una lista sencilla con solamente unas pocas operaciones asociadas para que la exposición no sea demasiado larga. En la práctica, las clases de objetos que definen una lista podrían tener probablemente más operaciones.

Supongamos que la primera etapa del proceso de especificación, es decir, la estructura de la especificación, se ha llevado a cabo y se ha identificado la necesidad de una lista. El nombre de la especificación y el nombre de la clase pueden ser el mismo, si bien es útil distinguir entre éstos usando alguna convención. Aquí se usan las mayúsculas para el nombre de la especificación (**LIST**) y minúsculas con la inicial en mayúsculas para el nombre de la clase (**List**). Como las listas son colecciones de otros tipos, la especificación tiene un parámetro genérico (**Elem**). El nombre **Elem** puede representar cualquier tipo: integer, string, list, y otros.

En general, para cada tipo abstracto de datos, las operaciones requeridas deberían incluir una operación para crear instancias de ese tipo (**Create**) y para construir el tipo a partir de sus elementos básicos (**Cons**). En el caso de las listas, debería haber una operación para evaluar el primer elemento de la lista (**Head**), una operación que devuelva la lista creada eliminando el primer elemento (**Tail**), y una operación para contar el número de elementos de la lista (**Length**).

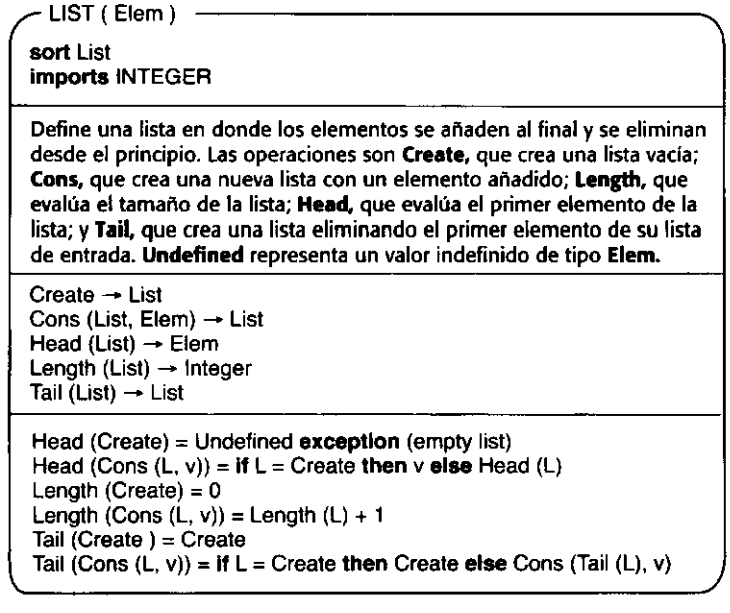


Figura 10.7 Una especificación de una lista sencilla.

Para definir la sintaxis de cada una de estas operaciones, debe decidirse qué parámetros se requieren para cada operación y los resultados de la operación. En general, los parámetros de entrada son tanto la clase que se está definiendo (**List**) como la clase genérica (**Elem**). Los resultados de las operaciones pueden ser objetos de esas clases o de alguna otra clase, como por ejemplo **Integer** o **Boolean**. En el ejemplo de la lista, la operación **Length** debería devolver un entero. Por lo tanto, se debe incluir una declaración «imports» declarando que la especificación de un entero se utiliza en la especificación de la lista.

Para crear la especificación, se define un conjunto de axiomas que se aplican al tipo abstracto y éstos especifican su semántica. Se definen los axiomas que utilizan las operaciones definidas en la parte de la signatura. Estos axiomas especifican la semántica indicando lo que es siempre cierto acerca del comportamiento de las entidades de ese tipo abstracto.

Las operaciones de un tipo abstracto de datos normalmente se clasifican en dos categorías:

1. *Operaciones de construcción* (constructores), que crean o modifican las entidades de la clase definidas en la especificación. Normalmente, estas operaciones reciben nombres como **Create**, **Update**, **Add** o, en este caso, **Cons**, que significa construir.
2. *Operaciones de inspección*, que evalúan los atributos de la clase definidos en la especificación. Normalmente, éstos tienen nombres como **Eval** o **Get**.

Una buena regla para redactar una especificación algebraica es establecer las operaciones de construcción y escribir un axioma para cada operación de inspección sobre cada constructor. Esto sugiere que si hay **m** operaciones de construcción y **n** operaciones de inspección, deberían definirse **m * n** axiomas.

Sin embargo, las operaciones de construcción asociadas con un tipo abstracto pueden no ser siempre constructores primitivos. Esto es, pueden definirse usando otros constructores y operaciones de inspección. Si se define una operación de construcción usando otros constructores, entonces solamente se necesita definir las operaciones de inspección usando los constructores primitivos.

En la especificación de la lista, las operaciones de construcción que construyen las listas son **Create**, **Cons** y **Tail**. Las operaciones de inspección son **Head** (devuelve el valor del primer elemento de la lista) y **Length** (devuelve el número de elementos de la lista), los cuales se usan para descubrir atributos de la lista. La operación **Tail**, sin embargo, no es un constructor primitivo. Por lo tanto, no es necesario definir axiomas sobre la operación **Tail** para las operaciones **Head** y **Length**, pero se tiene que definir **Tail** usando las operaciones de los constructores primitivos.

Evaluar el primer elemento de una lista vacía devuelve un valor indefinido. Las especificaciones de **Head** y **Tail** muestran que **Head** evalúa el principio de la lista y **Tail** evalúa la lista de entrada sin su primer elemento. La especificación de **Head** establece que el primer elemento de una lista creada mediante **Cons** es o bien el valor añadido a la lista (si la lista inicial está vacía) o el mismo que el primer elemento de la lista de parámetros inicial de la operación **Cons**. Añadir un elemento a una lista no afecta a su primer elemento, a menos que la lista esté vacía.

Normalmente se usa recursividad cuando se redactan especificaciones algebraicas. El valor de la operación **Tail** es la lista formada tomando la lista de entrada y eliminando su primer elemento. La definición de **Tail** muestra cómo la recursividad se usa en la construcción de especificaciones algebraicas. La operación se define sobre listas vacías, y después de manera recursiva sobre listas no vacías, terminando la recursividad cuando se obtenga una lista vacía.

Algunas veces es más fácil comprender las especificaciones recursivas desarrollando un pequeño ejemplo. Supongamos que tenemos una lista [5, 7] en donde 5 es el primer elemento de la lista y 7 es el final de la lista. La operación **Cons** ([5, 7], 9) debería devolver la lista [5, 7, 9] y la operación **Tail** aplicada a ésta debería devolver la lista [7, 9]. La secuencia de ecuaciones resultantes de sustituir los parámetros en la especificación anterior con estos valores es:

```
Tail ([5, 7, 9]) =
Tail (Cons ( [5, 7], 9)) = Cons (Tail [5, 7]), 9) =
        Cons (Tail (Cons ([5], 7)), 9) = Cons (Cons (Tail ([5]), 7), 9) =
        Cons (Cons (Tail (Cons ([], 5)), 7), 9) = Cons (Cons ([Create], 7), 9) =
            Cons ([7], 9) = [7, 9]
```

La reescritura sistemática del axioma para **Tail** ilustra que realmente produce el resultado esperado. Se puede probar que el axioma para **Head** es correcto usando la misma técnica de reescritura.

Ahora vamos a mostrar cómo se puede usar la especificación algebraica de una interfaz en una especificación de un sistema crítico. Supóngase que, en un sistema de control de tráfico aéreo, se ha diseñado un objeto que representa un sector controlado del espacio aéreo. Cada sector controlado puede incluir un número de avión, cada uno de los cuales tiene un único identificador de avión. Por razones de seguridad, todos los aviones deben separarse como mínimo 300 metros de altura. El sistema avisa al controlador si un avión intenta posicionarse de forma que incumpla esta restricción.

Para simplificar la descripción, solamente se ha definido un número limitado de operaciones sobre el objeto sector. En un sistema real, probablemente existan muchas más operaciones y muchas más condiciones de seguridad complejas relacionadas con la separación horizontal del avión. Las operaciones críticas sobre el objeto son:

1. **Enter**. Esta operación añade un avión (representado por un identificador) al espacio aéreo a una altura específica. No debe haber otro avión a esa altura o a menos de 300 metros de él.
2. **Leave**. Esta operación elimina el avión especificado del sector controlado. Esta operación se usa cuando el avión se traslada a un sector adyacente.
3. **Move**. Esta operación mueve un avión de una altura a otra. De nuevo, debe comprobarse la restricción de seguridad sobre la separación vertical a menos de 300 metros.
4. **Lookup**. Dado un identificador de avión, esta operación devuelve la altura actual del avión en el sector.

Es más fácil especificar estas operaciones si se definen algunas otras operaciones de la interfaz. Éstas son:

1. **Create**. Es una operación estándar para un tipo abstracto de datos. Provoca que se cree una instancia vacía del tipo. En este caso, ésta representa un sector que no tiene aviones.
2. **Put**. Es una versión más simple de la operación **Enter**. Añade un avión al sector sin chequear ninguna restricción asociada.
3. **In-space**. Dada una señal de llamada de un avión, esta operación de tipo lógico devuelve cierto si el avión está en el sector controlado, y falso en cualquier otro caso.
4. **Occupied**. Dada una altura, esta operación de tipo lógico devuelve cierto si hay un avión a menos de 300 metros de esa altura, y falso en cualquier otro caso.

La ventaja de definir estas operaciones más sencillas es que pueden entonces usarse como bloques de construcción para definir operaciones más complejas sobre la clase **Sector**. La especificación algebraica de esta clase se muestra en la Figura 10.8.

Fundamentalmente, las operaciones de construcción básicas son **Create** y **Put**, y aquí se usan en la especificación de otras operaciones. **Occupied** e **In-space** son operaciones de comprobación que se han definido utilizando **Create** y **Put**, y luego se usan en otras especifica-

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter – añade un avión al sector si se cumplen las condiciones de seguridad
Leave – elimina un avión en el sector
Move – mueve un avión de una altura a otra si es seguro hacerlo
Lookup – encuentra la altura de un avión en el sector

Create – crea un sector vacío
Put – añade un avión a un sector sin comprobaciones de restricciones
In-space – comprueba si un avión ya está en el sector
Occupied – comprueba si está disponible una altura especificada

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Enter (S, CS, H) =
  if In-space (S, CS) **then** S **exception** (Aircraft already in sector)
  elsif Occupied (S, H) **then** S **exception** (Height conflict)
  else Put (S, CS, H)

Leave (Create, CS) = Create **exception** (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** S **else** Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
  if S = Create **then** Create **exception** (No aircraft in sector)
  elsif **not** In-space (S, CS) **then** S **exception** (Aircraft not in sector)
  elsif Occupied (S, H) **then** S **exception** (Height conflict)
  else Put (Leave (S, CS), CS, H)

-- NO-HEIGHT es una constante que indica que no se puede devolver una altura válida

Lookup (Create, CS) = NO-HEIGHT **exception** (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** H1 **else** Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
  if (H1 > H **and** H1 - H ≤ 300) **or** (H > H1 **and** H - H1 ≤ 300) **then** true
  else Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS) =
  if CS = CS1 **then** true **else** In-space (S, CS)

Figura 10.8
Especificación de un sector controlado.

ciones. No se dispone de espacio para explicar aquí con detalle todas las operaciones pero se describen dos de ellas (**Occupied** y **Move**). Con esta información, usted debería ser capaz de comprender las especificaciones del resto de las operaciones.

1. La operación **Occupied** toma un sector como parámetro que representa la altura y comprueba si no hay ningún avión que tenga asignada dicha altura. Su especificación establece que:
   - En un sector vacío (uno que ha sido creado con la operación **Create**), cada nivel está vacante. La operación devuelve falso independientemente del valor del parámetro altura.
   - En un sector no vacío (uno sobre el que se han hecho previamente operaciones **Put**), la operación **Occupied** comprueba si la altura especificada (parámetro **H**) está a menos de 300 metros de la altura del avión que fue añadido en último lugar por una operación **Put**. Si es así, la altura ya está ocupada y, por tanto, el valor de **Occupied** es cierto.
   - Si el sector no está ocupado, la operación comprueba dicho sector de forma recursiva. Se puede pensar que esta operación se lleva a cabo sobre el último avión añadido en el sector. Si la altura no está dentro del rango de la altura de ese avión, la operación entonces comprueba con el avión anterior que ha sido añadido en el sector y así sucesivamente. Eventualmente, si no existe ningún avión en el rango de la altura especificada, la comprobación se lleva a cabo con un sector vacío y, por tanto, devuelve falso.
2. La operación **Move** mueve un avión de un sector desde una altura a otra. Su especificación establece que:
   - Si una operación **Move** se aplica sobre un espacio aéreo vacío (el resultado de **Create**), el espacio aéreo no cambia y se lanza una excepción para indicar que el avión especificado no está en el espacio aéreo.
   - En un sector no vacío, la operación primero comprueba (utilizando **In-space**) si el avión dado está en el sector. Si no, se lanza una excepción. Si está, la operación comprueba que la altura especificada esté disponible (utilizando **Occupied**), lanzando una excepción si ya hay un avión a esa altura.
   - Si la altura especificada está disponible, la operación **Move** es equivalente a que el avión especificado deje el espacio aéreo (por lo tanto, se usa la operación **Leave**) y se añada al sector a la nueva altura.

10.3 Especificación del comportamiento

Las sencillas técnicas algebraicas descritas en la sección anterior pueden usarse para describir interfaces en las que las operaciones del objeto son independientes de su estado. Es decir, los resultados de aplicar una operación no deberían depender de los resultados de operaciones anteriores. En caso de que estas condiciones no se cumplan, las técnicas algebraicas pueden resultar engorrosas. Además, a medida que éstas aumentan su tamaño, se observa que las descripciones algebraicas del comportamiento del sistema se vuelven más difíciles de entender.

Una aproximación alternativa a la especificación formal que se está utilizando cada vez más en proyectos industriales es la especificación basada en modelos. La especificación ba-

sada en modelos es una aproximación a la especificación formal en la que la especificación del sistema se expresa como un modelo de estados del sistema. Se pueden especificar las operaciones del sistema definiendo cómo éstas afectan al estado del modelo del sistema. La combinación de estas especificaciones define el comportamiento del sistema en su totalidad.

Algunas notaciones maduras para desarrollar especificaciones basadas en modelos son VDM (Jones, 1980; Jones, 1986), B (Wordsworth, 1996) y Z (Hayes, 1987; Spivey, 1992). En este libro se usa Z. En Z, los sistemas se modelan usando conjuntos y relaciones entre conjuntos. Sin embargo, Z ha ampliado estos conceptos matemáticos con construcciones que soportan de forma específica la especificación del software.

En una introducción a la especificación basada en modelos, solamente se puede proporcionar un vistazo general de cómo se puede desarrollar una especificación. Una descripción completa de la notación Z está fuera de este capítulo. En su lugar, se presentan algunos pequeños ejemplos para ilustrar la técnica e introducir la notación a medida que se requiera. Una completa descripción de la notación Z se encuentra en libros de texto tales como los de Diller (Potter *et al.*, 1996) y Jacky (Jacky, 1997).

Las especificaciones formales pueden ser difíciles y tediosas de leer, especialmente cuando se presentan como largas fórmulas matemáticas. Los diseñadores de Z han prestado una atención particular a este problema. Las especificaciones se presentan como texto informal complementado con descripciones formales. La descripción formal se incluye como pequeños trozos fáciles de leer (denominados *esquemas*) que se distinguen del texto asociado resaltándola gráficamente. Los esquemas se usan para introducir variables de estado y para definir restricciones y operaciones sobre ese estado. Los esquemas pueden ser manipulados utilizando operaciones tales como composición de esquemas, renombrado de esquemas y ocultación de esquemas.

Para ser más efectiva, una especificación formal debe complementarse con descripciones informales. La presentación del esquema Z se ha diseñado para que se destaque del texto que lo rodea (Figura 10.9).

La signatura del esquema define las entidades que forman el estado del sistema y el predicado del esquema establece las condiciones que deberían cumplirse siempre para estas entidades. Cuando un esquema define una operación, el predicado puede establecer precondiciones y postcondiciones. Éstas definen el estado antes y después de la operación. La diferencia entre estas pre y postcondiciones define la acción especificada en el esquema de la operación.

Para ilustrar el uso de Z en la especificación de un sistema crítico, se ha desarrollado una especificación formal del sistema de control de la bomba de insulina que se introdujo en el Capítulo 3.

Recuerde que este sistema monitoriza el nivel de glucosa en la sangre de los diabéticos y automáticamente inyecta la insulina necesaria. Incluso para un sistema pequeño como el de

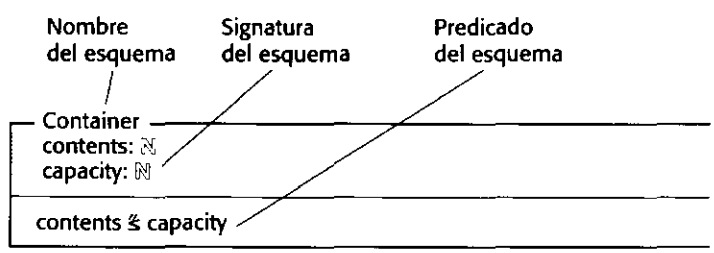


Figura 10.9
La estructura de un esquema Z.

la bomba de insulina, la especificación formal es bastante larga. Si bien la operación básica del sistema es sencilla, hay muchas condiciones de alarma posibles que tienen que ser consideradas. Aquí solamente se incluyen algunos de los esquemas que definen el sistema; la especificación completa puede descargarse del sitio web del libro.

Para desarrollar una especificación basada en modelos, se tienen que definir variables de estado y predicados que modelen el estado del sistema que se está especificando, así como definir invariantes (condiciones que son siempre ciertas) sobre esas variables de estado.

El esquema de estados Z que modela el estado de la bomba de insulina se muestra en la Figura 10.10. Puede verse cómo se usan las dos partes básicas del esquema. En la parte superior, se declaran los nombres y los tipos, y en la parte inferior, los invariantes.

Los nombres declarados en el esquema se usan para representar entradas del sistema, salidas del sistema y estados internos de las variables:

1. *Entradas del sistema* para las que la convención en Z es que todos los nombres de variables de entrada sean seguidos por el símbolo ?. Se han declarado nombres para modelar el interruptor encendido/apagado de la bomba (**switch?**), un botón para suministro manual de insulina (**ManualDeliveryButton?**), la lectura del sensor de azúcar en la sangre (**Reading?**), el resultado de ejecutar un programa de prueba del hardware (**HardwareTest?**), sensores que detectan el depósito de insulina y la aguja (**InsulinReservoir?**, **Needle?**), y el valor del tiempo actual (**clock?**).
2. *Salidas del sistema* para las que la convención en Z es que todos los nombres de variables sean seguidos por el símbolo !. Se han declarado nombres para modelar la alarma de la bomba (**alarm!**), dos pantallas alfanuméricas (**display1!** y **display2!**), una pantalla para el tiempo actual (**clock!**), y la dosis de insulina a suministrar (**dose!**).
3. *Variables de estado usadas para el cálculo de la dosis*. Se han declarado variables para representar el estado del dispositivo (**status**) para almacenar valores previos del nivel de azúcar en la sangre (**r0**, **r1** y **r2**), la capacidad del depósito de insulina y la capacidad de insulina actualmente disponible (**capacity**, **insulin_available**), varias variables usadas para limitar la dosis de insulina suministrada (**max_daily_dose**, **max_single_dose**, **minimun_dose**, **safemin**, **safemax**), y dos variables usadas en el cálculo de la dosis (**CompDose** y **cumulative_dose**). El tipo $\mathbb{N}$ significa un número no negativo.

El predicado del esquema define invariantes que son siempre ciertos. Aquí hay un «and» implícito entre cada línea del predicado, de forma que todos los predicados deben cumplirse durante todo el tiempo. Algunos de estos predicados simplemente fijan los límites del sistema, pero otros definen condiciones de funcionamiento fundamentales del sistema. Éstas comprenden las siguientes:

1. La dosis debe ser menor o igual que la capacidad del depósito de insulina. Es decir, es imposible suministrar más insulina de la que hay en el depósito.
2. La dosis acumulada se inicializa cada día a media noche. Se puede considerar que la frase en Z <expresión lógica 1> ⇒ <expresión lógica 2> significa lo mismo que **si** <expresión lógica 1> **entonces** <expresión lógica 2>. En este caso, <expresión lógica 1> es «**clock?=000000**» y <expresión lógica 2> es «**cumulative_dose=0**».
3. La dosis acumulada suministrada durante un periodo de 24 horas no puede superar **max_daily_dose**. Si esta condición es falsa, entonces se muestra un mensaje de error.
4. **display2!** siempre muestra el valor de la última dosis de insulina suministrada y **clock!** siempre muestra la hora actual.

INSULIN_PUMP_STATE

// Definición de dispositivo de entrada
switch?: (off, manual, auto)
ManualDeliveryButton?: ℕ
Reading?: ℕ
HardwareTest?: (OK, batterylow, pumpfail, sensorfail, deliveryfail)
InsulinReservoir?: (present, notpresent)
Needle?: (present, notpresent)
clock?: TIME

// Definición de dispositivo de salida
alarm! = (on, off)
display1!, string
display2!: string
clock!: TIME
dose!: ℕ

// Variables de estado usadas para el cálculo de la dosis
status: (running, warning, error)
r0, r1, r2: ℕ
capacity, insulin_available : ℕ
max_daily_dose, max_single_dose, minimum_dose: ℕ
safemin, safemax: ℕ
CompDose, cumulative_dose: ℕ

---

r2 = Reading?
dose! ≤ insulin_available
insulin_available ≤ capacity

// La dosis de insulina acumulada suministrada se inicializa a cero una vez cada 24 horas
clock? = 000000 ⇒ cumulative_dose = 0

// Si la dosis acumulada excede el límite entonces la operación se suspende
cumulative_dose ≥ max_daily_dose ∧ status = error ⇒
display1! = "Daily dose exceeded"

// Parámetros de configuración de la bomba
capacity = 100 ∧ safemin = 6 ∧ safemax = 14
max_daily_dose = 25 ∧ max_single_dose = 4 ∧ minimum_dose = 1

display2! = nat_to_string (dose!)
clock! = clock?

Figura 10.10
Esquema de estados para la bomba de insulina.

La bomba de insulina funciona comprobando la glucosa en la sangre cada 10 minutos, y (de forma muy simple) la insulina se suministra si se incrementa la velocidad a la que cambia la glucosa en la sangre. El esquema que hemos denominado RUN, mostrado en la Figura 10.11, modela la condición de funcionamiento normal de la bomba. Si el nombre de un esquema se incluye en la parte de declaraciones, esto es equivalente a incluir todos los nombres declarados en ese esquema en la declaración y las condiciones en la parte de predicados. El esquema delta (Δ) en la primera línea de la Figura 10.11 ilustra esto. El símbolo delta significa que las variables de estado definidas en **INSULIN_PUMP_STATE** pertenecen a su ámbito en tanto que son un conjunto de variables que representan valores de estado antes y después de la misma operación. Éstos se indican añadiendo una comilla simple al nombre definido en **INSULIN_PUMP_STATE**. Por lo tanto, **insulin_available** representa la cantidad de insulina

RUN

ΔINSULIN_PUMP_STATE

---

switch? = auto _
status = running ∨ status = warning
insulin_available ≥ max_single_dose
cumulative_dose < max_daily_dose

// La dosis de insulina se calcula dependiendo del nivel de azúcar en la sangre

(SUGAR_LOW ∨ SUGAR_OK ∨ SUGAR_HIGH)

// 1. Si la dosis calculada de insulina es cero, no suministre insulina

CompDose = 0 ⇒ dose! = 0

∨

// 2. La dosis diaria máxima podría sobrepasarse si se suministrase la dosis calculada, por lo que la dosis de insulina se calcula como la diferencia entre la dosis diaria máxima permitida y la dosis acumulada suministrada hasta ahora

CompDose + cumulative_dose > max_daily_dose ⇒ alarm! = on ∧ status' = warning ∧
dose! = max_daily_dose–cumulative_dose

∨

// 3. La situación normal. Si la dosis única máxima no es excedida, entonces suministrar la dosis calculada. Si la dosis única calculada es demasiado alta, restringir la dosis suministrada a la dosis única máxima

CompDose + cumulative_dose < max_daily_dose ⇒
(CompDose ≤ max_single_dose ⇒ dose! = CompDose
∨
CompDose > max_single_dose ⇒ dose! = max_single_dose)
insulin_available' = insulin_available–dose!
cumulative_dose' = cumulative_dose + dose!

insulin_available ≤ max_single_dose * 4 ⇒ status' = warning ∧
display1! = "Insulin low"

r1' = r2
r0' = r1

Figura 10.11
El esquema RUN.

disponible antes de alguna operación, e **insulin_available'** representa la cantidad de insulina disponible después de alguna operación.

El esquema **RUN** define el funcionamiento del sistema especificando un conjunto de predicados que son ciertos en el uso normal del sistema. Por supuesto, además de éstos, están los predicados definidos en el esquema **INSULIN_PUMP_STATE** que son invariantes (siempre ciertos). Este esquema también muestra el uso de una característica del lenguaje Z —composición de esquemas— en donde los esquemas **SUGAR_LOW**, **SUGAR_OK**, y **SUGAR_HIGH** se incluyen dando sus nombres. Tenga en cuenta que estos tres esquemas son «excluyentes» de forma que hay un esquema para cada una de las tres posibles condiciones. La capacidad para componer esquemas significa que se puede dividir una especificación en partes más pequeñas de la misma forma que se pueden definir funciones y métodos en un programa.

Aquí no vamos a entrar en detalles en el esquema RUN, pero, básicamente, comienza definiendo predicados que son ciertos en el funcionamiento normal. Por ejemplo, establece que el funcionamiento normal solamente es posible cuando la cantidad de insulina disponible es mayor que la dosis máxima única que puede suministrarse. Tres esquemas que representan diferentes niveles de azúcar en la sangre son por lo tanto excluyentes y, como veremos más adelante, éstos definen un valor para la variable de estado **Comp_Dose**.

El valor de **Comp_Dose** representa la cantidad de insulina que ha sido calculada para su suministro, basada en el nivel de azúcar en la sangre. El resto de los predicados en este esquema definen varias comprobaciones que deben ser aplicadas para asegurar que la dosis realmente suministrada (**dose!**) cumple las reglas de seguridad definidas para el sistema. Por ejemplo, una regla de seguridad es que ninguna dosis única de insulina pueda sobrepasar un valor máximo definido.

Finalmente, los dos últimos predicados definen los cambios para el valor de **insulin_available** y **cumulative_dose**. Observe cómo se ha usado aquí la versión con comilla simple.

El último ejemplo de esquema dado en la Figura 10.12 define cómo se calcula la dosis de insulina asumiendo que el nivel de azúcar en la sangre del diabético se encuentra en una zona

SUGAR_OK

$r2 \geq safemin \lor r2 \land safemax$

// Nivel de azúcar estable o en descenso

$r2 \leq r1 \Rightarrow CompDose = 0$

$\lor$

// Nivel de azúcar creciente pero velocidad de incremento decreciente

$r2 > r1 \land (r2\text{-}r1) < (r1\text{-}r0) \Rightarrow CompDose = 0$

$\lor$

// Cálculo de la dosis con nivel de azúcar creciente y velocidad de incremento creciente
// Se debe suministrar una dosis mínima si se puede redondear a cero

$r2 > r1 \land (r2\text{-}r1) \geq (r1\text{-}r0) \land (round\ ((r2\text{-}r1)/4) = 0) \Rightarrow$
$CompDose = minimum\_dose$

$\lor$

$r2 > r1 \land (r2\text{-}r1) \geq (r1\text{-}r0) \land (round\ ((r2\text{-}r1)/4) > 0) \Rightarrow$
$CompDose = round\ ((r2\text{-}r1)/4)$

Figura 10.12 El esquema SUGAR_OK.

de seguridad. En esas circunstancias, la insulina solamente se suministra si el nivel de azúcar en la sangre crece y también se incrementa la velocidad de cambio de azúcar en la sangre. El resto de los esquemas, SUGAR_LOW y SUGAR_HIGH definen la dosis a suministrar si el nivel de azúcar está fuera de la zona de seguridad. Los predicados en el esquema son los siguientes:

1. El predicado inicial define la zona de seguridad; es decir, r2 debe estar entre safemin y safemax.
2. Si el nivel de azúcar es estable o decreciente, indicado por r2 (la última lectura) siendo igual o menor que r1 (una lectura anterior), entonces la dosis de insulina a suministrar es cero.
3. Si el nivel de azúcar es creciente (r2 mayor que r1) pero la velocidad de crecimiento es decreciente, entonces la dosis a suministrar es cero.
4. Si el nivel de azúcar es creciente y la velocidad de crecimiento es estable, entonces se suministra una dosis mínima de insulina.
5. Si el nivel de azúcar es creciente y la velocidad de crecimiento es decreciente, entonces la velocidad de insulina a suministrar se calcula aplicando una sencilla fórmula a los valores calculados.

No modelamos el comportamiento temporal del sistema (por ejemplo, el hecho de que el sensor de glucosa se comprueba cada 10 minutos) usando el lenguaje Z. Si bien es ciertamente posible, también es engorroso, y podríamos decir una descripción informal realmente muestra la especificación de forma más concisa que una especificación formal.

PUNTOS CLAVE

- **Los métodos de especificación formal de sistemas complementan a las técnicas de especificación informal de requerimientos. Pueden utilizarse con la definición de requerimientos mediante lenguaje natural para clarificar cualquier área de ambigüedad potencial en la especificación.**
- **Las especificaciones formales son precisas y no ambiguas. Eliminan las áreas dudosas en una especificación y evitan algunos de los problemas de mala interpretación del lenguaje. Sin embargo, los no especialistas pueden encontrar estas especificaciones formales difíciles de entender.**
- **La ventaja fundamental de usar métodos formales en el proceso del software es que fuerza a un análisis de los requerimientos del sistema en una etapa inicial. Corregir errores en esta etapa es menos costoso que modificar un sistema ya entregado.**
- **Las técnicas de especificación formal son más rentables en el desarrollo de sistemas críticos en los que la seguridad, fiabilidad y protección son particularmente importantes. También pueden utilizarse para especificar estándares.**
- **Las técnicas algebraicas de especificación formal son especialmente adecuadas para especificar interfaces en donde la interfaz se define como un conjunto de clases de objetos o tipos abstractos de datos. Estas técnicas ocultan el estado del sistema y especifican el sistema en función de las relaciones entre las operaciones de la interfaz.**

- **Las técnicas basadas en modelos modelan el sistema utilizando construcciones matemáticas tales como conjuntos y funciones. Éstas pueden mostrar el estado del sistema, lo que simplifica algunos tipos de especificación del comportamiento.**
- **Las operaciones se definen en una especificación basada en modelos definiendo las precondiciones y postcondiciones sobre el estado del sistema.**

LECTURAS ADICIONALES

«Correctness by construction: Developing a commercially secure system». Una buena descripción de cómo pueden usarse los métodos formales en el desarrollo de un sistema de protección crítico. [A. Hall y R. Chapman, *IEEE Software*, 19(1), enero 2002.]

IEEE Transactions on Software Engineering, enero 1998. Este número de la revista incluye una sección especial sobre usos prácticos de los métodos formales en ingeniería del software. Incluye artículos del lenguaje Z y del lenguaje LARCH.

«Formal methods: Promises and problems». Este artículo es un análisis realista de los beneficios potenciales del uso de métodos formales y de las dificultades de integrar el uso de estos métodos en el desarrollo práctico del software. [Luqi y J. Goguen. *IEEE Software*, 14(1), enero 1997.]

EJERCICIOS

10.1 Sugiera por qué el diseño arquitectónico de un sistema debería preceder al desarrollo de una especificación formal.

10.2 Se le ha asignado la tarea de «vender» técnicas de especificación formal a una empresa de desarrollo de software. Indique cómo podría explicar las ventajas de las especificaciones formales a ingenieros de software escépticos y prácticos.

10.3 Explique por qué es particularmente importante definir las interfaces de los subsistemas de forma precisa y por qué la especificación algebraica es particularmente adecuada para la especificación de estas interfaces.

10.4 Un tipo abstracto de datos que representa una pila tiene las siguientes operaciones asociadas:

New: Crea una pila.

Push: Añade un elemento en la cima de la pila.

Top: Obtiene el elemento de la cima de la pila.

Retract: Elimina el elemento de la cima de la pila y devuelve la pila modificada.

Empty: Cierto si no hay elementos en la pila.

Defina este tipo abstracto de datos utilizando una especificación algebraica.

10.5 En el ejemplo de un sector controlado del espacio aéreo, la condición de seguridad es que no debe estar dentro de los 300 m de altura en el mismo sector. Modifique la especificación mostrada en la Figura 10.8 para permitir que el avión ocupe la misma altura en el sector si está separado al menos 8 km horizontalmente. Usted puede prescindir de los aviones en los sectores adyacentes. *Sugerencia*: tiene que modificar las operaciones de construcción para que incluyan la posición del avión así como su altura. También tiene que definir una operación que, dadas dos posiciones, devuelva la separación entre ellas.

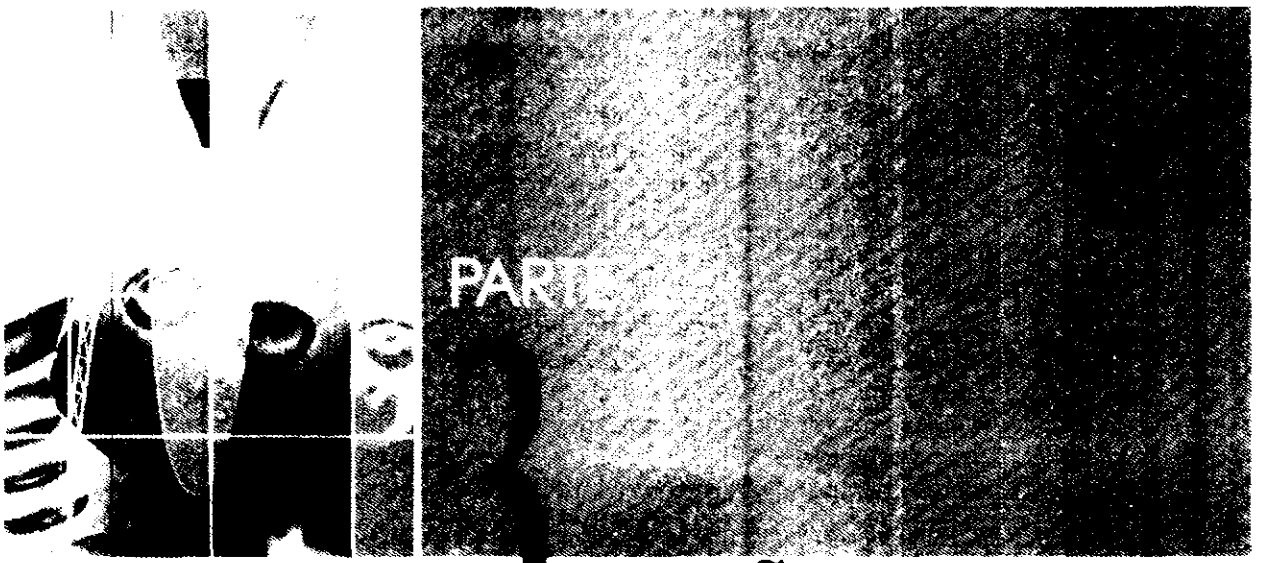
10.6 Los cajeros automáticos de los bancos utilizan la información de las tarjetas de los usuarios mediante el identificador del banco, el número de cuenta y el identificador personal del usuario. También generan información de la cuenta a partir de una base de datos central y actualizan dicha base de datos al completar la transacción. Empleando su conocimiento del funcionamiento de un ATM, escriba esquemas Z que definan el estado del sistema, la validación de la tarjeta (en la que se comprueba la identificación del usuario) y la retirada de efectivo.

10.7 Modifique el esquema de bomba de insulina, mostrado en la Figura 10.10, y añada una condición de seguridad adicional de forma que el ManualDeliveryButton? pueda tener solamente un valor distinto de cero si el interruptor de la bomba está en la posición manual.

10.8 Escriba un esquema Z denominado SELF_TEST que compruebe los componentes hardware de la bomba de insulina y fije los valores de la variable de estado HarwareTest?. A continuación modifique el esquema RUN para comprobar que el hardware funciona correctamente antes de que se suministre insulina. Si no, la dosis suministrada debería ser cero y se debería mostrarse un error en la pantalla de la bomba de insulina.

10.9 Z soporta la noción de sucesiones en donde una sucesión es como un vector. Por ejemplo, para una sucesión S, usted puede referirse a sus elementos como S[1], S[2], y así sucesivamente. También le permite determinar el número de elementos de una sucesión usando el operador #. Es decir, si una sucesión S es [a, b, c, d], entonces #S es 4. Usted puede añadir un elemento al final de una sucesión S escribiendo S + a, y al principio de la secuencia escribiendo a + S. Usando estas construcciones, escriba una especificación Z de la LIST que se especifica algebraicamente en la Figura 10.7.

10.10 Usted es un ingeniero de sistemas y se le pide que sugiera la mejor manera para desarrollar el software de un sistema de seguridad crítico para un marcapasos del corazón. Usted sugiere especificar formalmente el sistema, pero su gestor rechaza su sugerencia. Usted piensa que sus razones son débiles y basadas en prejuicios. ¿Es ético desarrollar el sistema usando métodos que usted piensa que son inadecuados?



PARTE 3

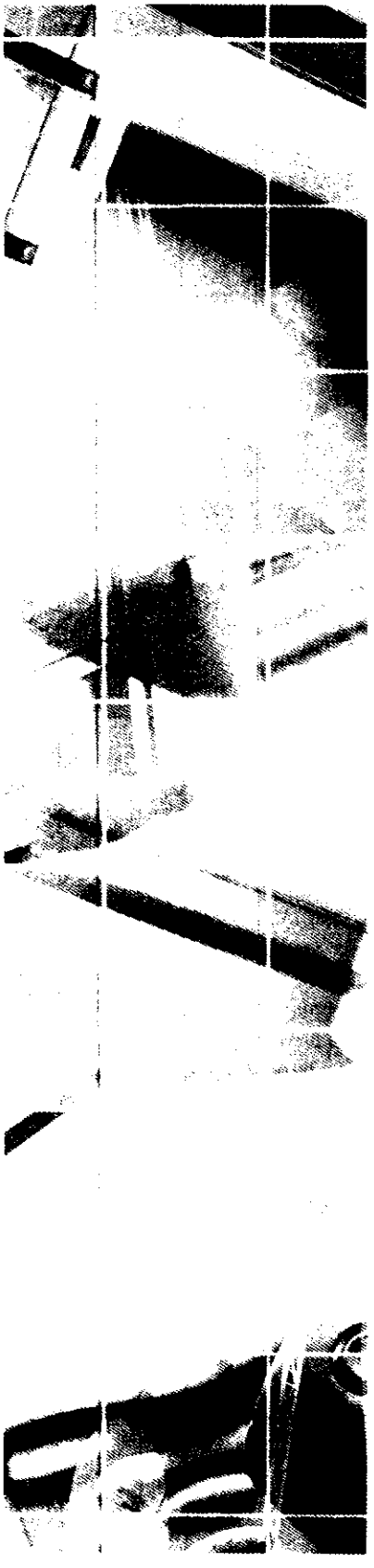
DISEÑO

La esencia del diseño del software es la toma de decisiones sobre la organización lógica del software. Algunas veces, usted representa esta organización lógica como un modelo en un lenguaje definido de modelado tal como UML y otras veces usted simplemente utiliza notaciones informales y esbozos para representar el diseño. Por supuesto, usted raramente empieza desde cero cuando toma decisiones sobre la organización del software sino que basa su diseño sobre experiencias de diseño anteriores.

Algunos autores piensan que la mejor forma de encapsular esta experiencia de diseño es mediante métodos estructurados en donde usted sigue un proceso definido de diseño y describe su diseño utilizando diferentes tipos de modelos. Yo nunca he sido un gran seguidor de los métodos estructurados ya que siempre he pensado que son demasiado restrictivos. El diseño es un proceso creativo y yo creo firmemente que cada uno de nosotros abordamos dicho proceso creativo de forma particular. No hay una forma buena o mala de diseñar el software y ni yo ni nadie puede darle una 'fórmula' para el diseño del software. Usted aprende cómo diseñar observando ejemplos de diseños existentes y discutiendo su diseño con otros.

En lugar de representar la experiencia como un «método de diseño», yo prefiero una aproximación menos estructurada. Los capítulos de esta parte encapsulan conocimiento sobre estructuras de software que han sido utilizadas con éxito en otros sistemas, presentan algunos ejemplos y le proporcionan algún consejo sobre procesos de diseño:

1. Los Capítulos del 11 al 13 hablan sobre las estructuras abstractas del software. El Capítulo 11 discute perspectivas estructurales que se han encontrado útiles para el diseño del software, el Capítulo 12 habla sobre cómo estructurar el software para su ejecución distribuida y el Capítulo 13 habla sobre estructuras genéricas para varios tipos de aplicaciones. El Capítulo 13 es nuevo y lo he incluido en esta edición debido a que he visto que muchos estudiantes de ingeniería del software no tienen experiencia en software de aplicaciones a parte de los sistemas interactivos que ellos usan de forma cotidiana en sus propios ordenadores.
2. Los Capítulos del 14 al 16 abordan cuestiones de diseño del software más específicas. El Capítulo 14, que cubre el diseño orientado a objetos, trata una forma de pensar sobre las estructuras del software. El Capítulo 15, dedicado al diseño de sistemas de tiempo real, discute las estructuras del software que usted necesita en sistemas en los que una respuesta a tiempo es un requerimiento crítico. El Capítulo 16 es un poco diferente debido a que está centrado en el diseño de la interfaz de usuario en vez de estructuras del software. Como ingeniero, usted tiene que pensar sobre los sistemas —no solamente sobre el software— y las personas del sistema son un componente esencial. El diseño no termina con la definición de las estructuras del software sino que continúa con cómo se usa el software.



11 Diseño arquitectónico

Objetivos

El objetivo de este capítulo es introducir los conceptos de la arquitectura del software y del diseño arquitectónico. Cuando haya leído este capítulo:

- comprenderá por qué es importante el diseño arquitectónico del software;
- comprenderá las decisiones que tienen que tomarse sobre la arquitectura del sistema durante el proceso de diseño arquitectónico;
- habrá sido introducido en tres estilos arquitectónicos complementarios que abarcan la organización del sistema en su totalidad, la descomposición modular y el control;
- comprenderá cómo se utilizan las arquitecturas de referencia para comunicar conceptos arquitectónicos y para evaluar las arquitecturas de los sistemas.

Contenidos

Los grandes sistemas siempre se descomponen en subsistemas que proporcionan algún conjunto de servicios relacionados. El proceso de diseño inicial que identifica estos subsistemas y establece un marco para el control y comunicación de los subsistemas se llama diseño arquitectónico. El resultado de este proceso de diseño es una descripción de la arquitectura del software.

En el modelo presentado en el Capítulo 4, el diseño arquitectónico es la primera etapa en el proceso de diseño y representa un enlace crítico entre los procesos de ingeniería de diseño y de requerimientos. El proceso de diseño arquitectónico está relacionado con el establecimiento de un marco estructural básico que identifica los principales componentes de un sistema y las comunicaciones entre estos componentes.

Bass y otros (Bass *et al.*, 2003) señalan tres ventajas de diseñar explícitamente y documentar la arquitectura del software:

1. *Comunicación con los stakeholders.* La arquitectura constituye una presentación de alto nivel del sistema que puede usarse como punto de discusión por varios *stakeholders*.
2. *Análisis del sistema.* Hacer explícita la arquitectura del sistema en una etapa temprana del desarrollo del sistema requiere realizar algún análisis. Las decisiones de diseño arquitectónico tienen un gran efecto sobre si el sistema puede cumplir los requerimientos críticos tales como rendimiento, fiabilidad y mantenibilidad.
3. *Reutilización a gran escala.* Un modelo de arquitectura del sistema es una descripción compacta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. La arquitectura del sistema es a menudo la misma para sistemas con requerimientos similares y, por lo tanto, pueden soportar reutilización del software a gran escala. Tal y como se indica en el Capítulo 18, es posible desarrollar arquitecturas para líneas de productos en donde la misma arquitectura se usa en varios sistemas relacionados.

Hofmeister y otros (Hofmeister *et al.*, 2000) ponen de manifiesto cómo las etapas del diseño arquitectónico fuerzan a los diseñadores del software a considerar aspectos de diseño claves en etapas tempranas del proceso. Sugieren que la arquitectura del software puede servir como un plan de diseño que se usa para negociar los requerimientos del sistema y como una forma de estructurar las discusiones con los clientes, desarrolladores y gestores. También sugieren que es una herramienta esencial para la gestión de la complejidad. La arquitectura del software oculta detalles y permite a los diseñadores centrarse en las abstracciones clave del sistema.

La arquitectura del sistema afecta al rendimiento, solidez, grado de distribución y mantenibilidad de un sistema (Bosch, 2000). El estilo y estructura particulares elegidos para una aplicación puede, por lo tanto, depender de los requerimientos no funcionales del sistema:

1. *Rendimiento.* Si el rendimiento es un requerimiento crítico, la arquitectura debería diseñarse para identificar las operaciones críticas en un pequeño número de subsistemas, con tan poca comunicación como sea posible entre estos subsistemas. Esto puede significar el uso relativo de componentes de grano grueso en vez de componentes de grano fino para reducir las comunicaciones entre ellos.
2. *Protección.* Si la protección es un requerimiento crítico, debería usarse una arquitectura estructurada en capas, con los recursos más críticos protegidos en las capas más internas y aplicando una validación de seguridad de alto nivel en dichas capas.
3. *Seguridad.* Si la seguridad es un requerimiento crítico, la arquitectura debería diseñarse para que las operaciones relacionadas con la seguridad se localizaran en un úni-

co subsistema o en un pequeño número de subsistemas. Esto reduce los costes y los problemas de validación de seguridad y hace posible crear los sistemas de protección relacionados con los de seguridad.

4. *Disponibilidad.* Si la disponibilidad es un requerimiento crítico, la arquitectura debería diseñarse para incluir componentes redundantes y para que sea posible reemplazar y actualizar componentes sin detener el sistema. Las arquitecturas de sistemas tolerantes a defectos para sistemas de alta disponibilidad se tratan en el Capítulo 20.
5. *Mantenibilidad.* Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debería diseñarse usando componentes independientes de grano fino que puedan modificarse con facilidad. Los productores de los datos deberían separarse de los consumidores y deberían evitarse las estructuras de datos compartidas.

Obviamente, existe un conflicto potencial entre algunas de estas arquitecturas. Por ejemplo, el uso de componentes de grano grueso mejora el rendimiento, y el uso de componentes de grano fino mejora la mantenibilidad. Si ambas propiedades son requerimientos importantes del sistema, entonces debería encontrarse una solución intermedia. Tal y como se indica más adelante, esto puede conseguirse en ocasiones usando diferentes estilos arquitectónicos para diferentes partes del sistema.

Hay un solape significativo entre los procesos de ingeniería de requerimientos y del diseño arquitectónico. Idealmente, una especificación del sistema no debería incluir ninguna información de diseño. En la práctica, esto no es realista excepto para sistemas muy pequeños. La descomposición arquitectónica es necesaria para estructurar y organizar la especificación. Un ejemplo de esto se introdujo en el Capítulo 2, en donde la Figura 2.8 muestra la arquitectura de un sistema de control de tráfico aéreo. Puede usarse dicho modelo arquitectónico como punto de partida para la especificación de subsistemas.

Un diseño de un subsistema es una descomposición abstracta de un sistema en componentes de grano grueso, cada uno de los cuales puede ser un sistema importante por sí mismo. Los diagramas de bloques se usan a menudo para describir diseños de subsistemas en donde cada caja en el diagrama representa un subsistema. Cajas dentro de otras cajas indican que el subsistema se ha descompuesto a su vez en otros subsistemas. Las flechas significan que los datos o señales de control pasan de un subsistema a otro subsistema en la dirección de las flechas. Los diagramas de bloques presentan un dibujo de alto nivel de la estructura del sistema, la cual puede entenderse con facilidad por personas de diferentes disciplinas que están implicadas en el proceso de desarrollo del sistema.

Por ejemplo, la Figura 11.1 es un modelo abstracto de arquitectura para un sistema robótico para empaquetar que muestra los subsistemas que tienen que desarrollarse. Este sistema robótico puede empaquetar diferentes clases de objetos. Utiliza un subsistema de visión para tomar los objetos de una cinta transportadora, identificar el objeto y seleccionar el tipo de empaquetado adecuado. A continuación, el sistema mueve los objetos desde la cinta transportadora para ser empaquetados. El sistema coloca los objetos empaquetados en otra cinta transportadora. Otros ejemplos de diseño arquitectónico a este nivel se muestran en el Capítulo 2 (Figuras 2.6 y 2.8).

Bass y otros (Bass *et al.*, 2003) señalan que los diagramas sencillos de cajas y líneas no son representaciones arquitectónicas útiles debido a que no muestran la naturaleza de las relaciones entre los componentes del sistema ni tampoco las propiedades externamente visibles de los componentes. Desde la perspectiva de un diseñador de software, esto es absolutamente correcto. Sin embargo, este tipo de modelo es efectivo para la comunicación con los *stake-*

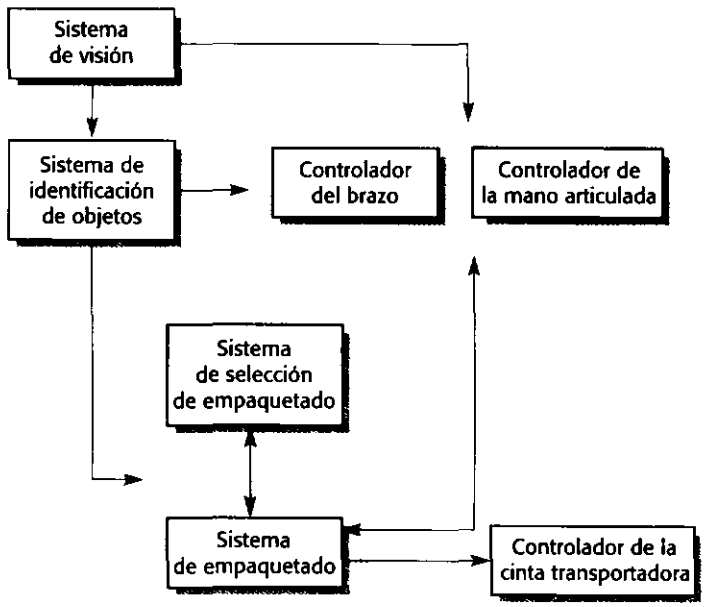


Figura 11.1 Diagrama de bloques de un sistema robótico de control de empaquetado.

holders del sistema y para la planificación del proyecto debido a que no muestra los detalles. Los *stakeholders* pueden relacionarlo con el sistema y tener una visión abstracta del mismo. El modelo identifica los subsistemas clave que tienen que ser desarrollados de forma independiente para que los gestores puedan comenzar a asignar personas para planificar el desarrollo de estos sistemas. Los diagramas de cajas y líneas, desde luego, no deberían ser la única representación arquitectónica usada; sin embargo, constituyen un modelo más de entre los modelos arquitectónicos que pueden resultar útiles.

El problema general de decidir cómo descomponer un sistema en subsistemas es difícil. Por supuesto, los requerimientos del sistema son un factor fundamental, y debería intentarse crear un diseño en el que hubiera una clara correspondencia entre los requerimientos y los subsistemas. Esto significa que, si los requerimientos cambian, este cambio probablemente esté localizado en un único sitio en vez de distribuido entre varios subsistemas. En el Capítulo 13, se describen varias arquitecturas de aplicaciones genéricas que pueden usarse como punto de partida para la identificación de subsistemas.

11.1 Decisiones de diseño arquitectónico

El diseño arquitectónico es un proceso creativo en el que se intenta establecer una organización del sistema que satisfaga los requerimientos funcionales y no funcionales del propio sistema. Debido a que es un proceso creativo, las actividades dentro del proceso difieren radicalmente dependiendo del tipo de sistema a desarrollar, el conocimiento y la experiencia del arquitecto del sistema, y los requerimientos específicos del mismo. Es, por tanto, más útil pensar en el proceso de diseño arquitectónico desde una perspectiva de decisión en lugar de una perspectiva de actividades. Durante el proceso de diseño arquitectónico, los arquitectos del sistema tienen que tomar varias decisiones fundamentales que afectan profundamente al sistema y a su proceso de desarrollo. Basándose en su conoci-

miento y experiencia, los arquitectos del sistema tienen que responder a las siguientes cuestiones fundamentales:

1. ¿Existe una arquitectura de aplicación genérica que pueda actuar como una plantilla para el sistema que se están diseñando?
2. ¿Cómo se distribuirá el sistema entre varios procesadores?
3. ¿Qué estilo o estilos arquitectónicos son apropiados para el sistema?
4. ¿Cuál será la aproximación fundamental utilizada para estructurar el sistema?
5. ¿Cómo se descompondrán en módulos las unidades estructurales del sistema?
6. ¿Qué estrategia se usará para controlar el funcionamiento de las unidades del sistema?
7. ¿Cómo se evaluará el diseño arquitectónico?
8. ¿Cómo debería documentarse la arquitectura del sistema?

Si bien cada sistema software es único, los sistemas en el mismo dominio de aplicación a menudo tienen arquitecturas similares que reflejan los conceptos fundamentales del dominio. Estas arquitecturas de las aplicaciones pueden ser bastante genéricas, tales como la arquitectura de los sistemas de gestión de información, o pueden ser mucho más específicas. Por ejemplo, las aplicaciones de líneas de productos son aplicaciones construidas sobre una arquitectura base con variantes que satisfacen los requerimientos específicos del cliente. Cuando se diseña la arquitectura de un sistema, se debe decidir qué tiene en común ese sistema con clases de aplicaciones más amplias, y determinar en qué medida el conocimiento de esas arquitecturas de aplicaciones se puede reutilizar. Las arquitecturas de las aplicaciones genéricas se estudian en el Capítulo 13, y en el Capítulo 18 se analizan las aplicaciones de líneas de productos.

Para sistemas embebidos y sistemas diseñados para computadoras personales, se utiliza normalmente un único procesador, y no será preciso diseñar una arquitectura distribuida para el sistema. Sin embargo, la mayoría de los sistemas grandes son actualmente sistemas distribuidos en los que el software del sistema se distribuye entre muchas computadoras diferentes. La elección de la arquitectura de distribución es una decisión clave que afecta al rendimiento y la fiabilidad del sistema. Ésta es una cuestión fundamental en sí misma, y este tema se trata por separado en el Capítulo 12.

La arquitectura de un sistema software puede basarse en un modelo o estilo arquitectónico particular. Un estilo arquitectónico es un patrón de organización de un sistema (Garlan y Shaw, 1993) tal como una organización cliente-servidor o una arquitectura por capas. Es importante un conocimiento de estos estilos, sus aplicaciones, y sus ventajas e inconvenientes. Sin embargo, las arquitecturas de la mayoría de los sistemas grandes no utilizan un único estilo. Pueden diseñarse diferentes partes del sistema utilizando distintos estilos arquitectónicos. En algunos casos, la totalidad de la arquitectura del sistema puede ser una arquitectura compuesta creada mediante la combinación de diferentes estilos arquitectónicos.

La noción de Garlan y Shaw de un estilo arquitectónico incluye las tres siguientes cuestiones de diseño. Se debe elegir la estructura más adecuada, tal como una estructura cliente-servidor o por capas, que permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural en módulos, hay que decidir la estrategia para descomponer subsistemas en sus componentes o módulos. Las aproximaciones que pueden usarse permiten implementar diferentes tipos de arquitecturas. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de los subsistemas. Se desarrolla un modelo general de las relaciones de control entre las partes establecidas del sistema. Estos tres temas se tratan en las Secciones de la 11.2 hasta la 11.4.

La evaluación de un diseño arquitectónico es difícil debido a que la verdadera prueba de una arquitectura consiste en averiguar el grado de satisfacción de sus requerimientos funcio-

nales y no funcionales después de que aquél ha sido desarrollado. Sin embargo, en algunos casos, se puede realizar alguna evaluación comparando el diseño elaborado con modelos arquitectónicos genéricos o de referencia. Las arquitecturas de referencia se tratan en la Sección 11.5, y otras arquitecturas genéricas, en el Capítulo 13.

El resultado del proceso de diseño arquitectónico es un documento de diseño arquitectónico. Éste puede incluir varias representaciones gráficas del sistema junto con texto descriptivo asociado. Debería describir cómo se estructura el sistema en subsistemas, la aproximación adoptada y cómo se estructura cada subsistema en módulos. Los modelos gráficos del sistema presentan diferentes perspectivas de la arquitectura. Los modelos arquitectónicos que pueden desarrollarse pueden incluir:

1. *Un modelo estructural estático* que muestre los subsistemas o componentes que han sido desarrollados como unidades separadas.
2. *Un modelo de proceso dinámico* que muestre cómo se organiza el sistema en procesos en tiempo de ejecución. Este modelo puede ser diferente del modelo estático.
3. *Un modelo de interfaz* que defina los servicios ofrecidos por cada subsistema a través de su interfaz pública.
4. *Modelos de relaciones* que muestren las relaciones, tales como el flujo de datos, entre los subsistemas.
5. *Un modelo de distribución*, que muestre cómo se distribuyen los subsistemas entre las computadoras.

Varios investigadores han propuesto el uso de lenguajes de descripción de arquitecturas (ADLs) para describir las arquitecturas de los sistemas. Bass y otros (Bass *et al.*, 2003) describen las principales características de estos lenguajes. Los elementos básicos de los ADLs son componentes y conectores, e incluyen reglas y recomendaciones para arquitecturas bien formadas. Sin embargo, como todos los lenguajes especializados, los ADLs solamente pueden ser comprendidos por expertos del lenguaje y son inaccesibles para los especialistas tanto de las aplicaciones como del dominio. Esto hace que sea difícil analizarlos desde una perspectiva práctica. Presumiblemente, sólo se usarán en unas pocas aplicaciones. Los modelos informales y las notaciones tales como UML (Clements, *et al.*, 2002) serán las notaciones más comúnmente usadas para la descripción arquitectónica.

11.2 Organización del sistema

La organización de un sistema refleja la estrategia básica usada para estructurar dicho sistema. Deben tomarse decisiones sobre la totalidad del modelo organizacional de un sistema al principio del proceso de diseño arquitectónico. La organización del sistema puede reflejarse directamente en la estructura de los subsistemas. Sin embargo, es frecuente que el modelo de subsistemas incluya más detalle que el organizacional, y no siempre hay una correspondencia sencilla desde los subsistemas a la estructura organizacional.

En esta sección, se incluyen tres estilos organizacionales ampliamente usados: un estilo de repositorio de datos, un estilo de servicios y servidores compartidos y una máquina abstracta o estilo por capas en donde el sistema se organiza en un conjunto de capas funcionales. Estos estilos se pueden utilizar juntos o por separado. Por ejemplo, un sistema puede organizarse alrededor de un repositorio de datos compartidos pero puede también construir capas alrededor de dicho repositorio para presentar una visión más abstracta de los datos.

11.2.1 El modelo de repositorio

Los subsistemas que forman un sistema deben intercambiar información para que puedan trabajar conjuntamente de forma efectiva. Esto es puede conseguir de dos formas fundamentales:

1. Todos los datos compartidos se almacenan en una base de datos central a la que puede acceder por todos los subsistemas. Un modelo de sistema basado en una base de datos compartida se denomina algunas veces *modelo de repositorio*.
2. Cada subsistema mantiene su propia base de datos. Los datos se intercambian con otros subsistemas mediante el paso de mensajes entre ellos.

La mayoría de los sistemas que usan grandes cantidades de datos se organizan alrededor de una base de datos compartida o repositorio. Por lo tanto, este modelo es adecuado para aplicaciones en las que los datos son generados por un subsistema y son usados por otro. Ejemplos de este tipo de sistemas se pueden citar los sistemas de mando y control, los sistemas de gestión de información, los sistemas CAD y los conjuntos de herramientas CASE.

La Figura 11.2 es un ejemplo de una arquitectura de herramientas CASE basada en un repositorio compartido. El primer repositorio compartido para herramientas CASE se desarrolló probablemente a principios de los 70 por una compañía inglesa denominada ICL para soportar el desarrollo de su sistema operativo (McGuffin *et al.*, 1979). Este modelo llegó a ser ampliamente conocido cuando Buxton (Buxton, 1980) hizo las propuestas para el entorno Stoneman como soporte para el desarrollo de sistemas escritos en Ada. Desde entonces, muchos de los conjuntos de herramientas CASE se han desarrollado alrededor de un repositorio compartido.

Las ventajas y desventajas de un repositorio compartido son las siguientes:

1. Es una forma eficiente de compartir grandes cantidades de datos. No hay necesidad de transmitir datos explícitamente de un subsistema a otro.
2. Sin embargo, los subsistemas deben estar acordes con el modelo de datos del repositorio. Inevitablemente, hay un compromiso entre las necesidades específicas de cada herramienta. El rendimiento puede verse afectado de forma adversa por este compromiso. Puede ser difícil o imposible integrar nuevos subsistemas si sus modelos de datos no se ajustan al esquema acordado.
3. Los subsistemas que producen datos no necesitan conocer cómo se utilizan sus datos por otros subsistemas.
4. Sin embargo, la evolución puede ser difícil a medida que se genera un gran volumen de información de acuerdo con el modelo de datos establecido. Traducir esto a un nuevo modelo, desde luego, tendrá un coste elevado; puede ser difícil o incluso imposible.

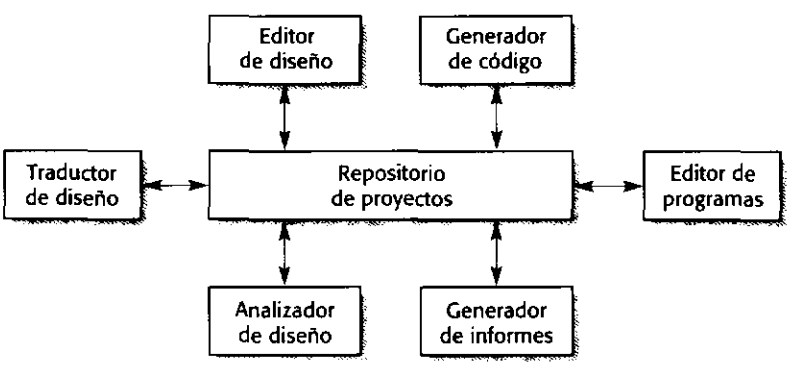


Figura 11.2 La arquitectura de un conjunto de herramientas CASE integradas.

5. Las actividades tales como copias de seguridad, protección, control de acceso y recuperación de errores están centralizadas. Son las responsables de gestionar el repositorio. Las herramientas pueden centrarse en su función principal en lugar de estar relacionadas con estas cuestiones.
6. Sin embargo, diferentes subsistemas pueden tener distintos requerimientos de protección, recuperación y políticas de seguridad. El modelo de repositorio impone la misma política para todos los subsistemas.
7. El modelo de compartición es visible a través del esquema del repositorio. Las nuevas herramientas se integran de forma directa puesto que éstas son compatibles con el modelo de datos acordado.
8. Sin embargo, puede ser difícil distribuir el repositorio sobre varias máquinas. Si bien es posible un repositorio centralizado lógicamente, puede haber problemas con la redundancia de datos y las inconsistencias.

En el modelo anterior, el repositorio es pasivo y el control es responsabilidad de los subsistemas que utilizan el repositorio. Una aproximación alternativa se deriva de los sistemas de AI (Inteligencia Artificial) que usan un modelo de pizarra (*blackboard*), el cual activa a los subsistemas cuando están disponibles ciertos datos. Esto es adecuado cuando el repositorio de datos está poco estructurado. Las decisiones sobre qué herramienta se tiene que activar sólo puede realizarse cuando los datos han sido analizados. Este modelo ha sido descrito por Nii (Nii, 1986) y Bosch (Bosch, 2000) e incluye una buena demostración sobre cómo este estilo se relaciona con los atributos de calidad del sistema.

11.2.2 El modelo cliente-servidor

El modelo arquitectónico cliente-servidor es un modelo de sistema en el que dicho sistema se organiza como un conjunto de servicios y servidores asociados, más unos clientes que acceden y usan los servicios. Los principales componentes de este modelo son:

1. Un conjunto de servidores que ofrecen servicios a otros subsistemas. Ejemplos de servidores son servidores de impresoras que ofrecen servicios de impresión, servidores de ficheros que ofrecen servicios de gestión de ficheros y servidores de compilación, que ofrecen servicios de compilación de lenguajes de programación.
2. Un conjunto de clientes que llaman a los servicios ofrecidos por los servidores. Éstos son normalmente subsistemas en sí mismos. Puede haber varias instancias de un programa cliente ejecutándose concurrentemente.
3. Una red que permite a los clientes acceder a estos servicios. Esto no es estrictamente necesario ya que los clientes y los servidores podrían ejecutarse sobre una única máquina. En la práctica, sin embargo, la mayoría de los sistemas cliente-servidor se implementan como sistemas distribuidos.

Los clientes pueden conocer los nombres de los servidores disponibles y los servicios que éstos proporcionan. Sin embargo, los servidores no necesitan conocer la identidad de los clientes o cuántos clientes tienen. Los clientes acceden a los servicios proporcionados por un servidor a través de llamadas a procedimientos remotos usando un protocolo de petición-respuesta tal como el protocolo http usado en la WWW. Básicamente, un cliente realiza una petición a un servidor y espera hasta que recibe una respuesta.

La Figura 11.3 muestra un ejemplo de un sistema basado en el modelo cliente-servidor. Éste es un sistema multiusuario basado en web para proporcionar una biblioteca de películas

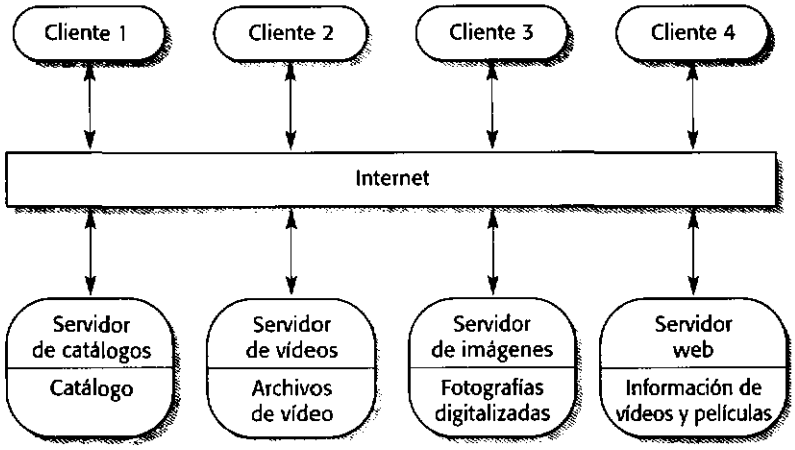


Figura 11.3 La arquitectura de un sistema de biblioteca de películas y fotografías.

y fotografías. En este sistema, varios servidores gestionan y visualizan diferentes tipos de dispositivos. Las secuencias de vídeo necesitan ser transmitidas rápidamente y en sincronía, pero con una resolución relativamente baja. Éstas pueden comprimirse en un almacén para que el servidor de vídeo pueda gestionar la compresión y descompresión de vídeo en diferentes formatos. Sin embargo, las fotografías deben mantenerse con una alta resolución, por lo que es adecuado mantenerlas en un servidor separado.

El catálogo debe ser capaz de manejar una gran variedad de peticiones y proporcionar enlaces al sistema de información web que incluye datos sobre las películas de vídeo y fotografías y un sistema de comercio electrónico que soporta la venta de películas de vídeo y fotografías. El programa cliente es simplemente una interfaz de usuario integrada con estos servicios y construida usando un navegador web.

La ventaja más importante del modelo cliente-servidor es que es una arquitectura distribuida. Se puede hacer un uso efectivo de los sistemas en red con muchos procesadores distribuidos. Es fácil añadir un nuevo servidor e integrarlo con el resto del sistema o actualizar los servidores de forma transparente sin afectar al resto del sistema. En el Capítulo 12 se estudian con más detalle las arquitecturas distribuidas, entre las que se encuentran las arquitecturas cliente-servidor y las arquitecturas de objetos distribuidos.

Sin embargo, puede ser necesario realizar cambios a los clientes y servidores existentes para obtener los mayores beneficios de la integración de un nuevo servidor. Puede no haber un modelo de datos compartidos entre los servidores, y los subsistemas pueden organizar sus datos de formas diferentes. Esto significa que los modelos de datos específicos pueden establecerse para cada servidor con el fin de optimizar su rendimiento. Por supuesto, si se usa una representación de los datos basada en XML, puede ser relativamente simple realizar conversiones de un esquema a otro. Sin embargo, XML es una forma ineficiente de representar los datos; por lo tanto, su uso puede provocar problemas de rendimiento.

11.2.3 El modelo de capas

El modelo de capas de una arquitectura (algunas veces denominada modelo de máquina abstracta) organiza el sistema en capas, cada una de las cuales proporciona un conjunto de servicios. Cada capa puede pensarse como una máquina abstracta cuyo lenguaje máquina se define por los servicios proporcionados por la capa. Este «lenguaje» se usa para implementar el siguiente nivel de la máquina abstracta. Por ejemplo, una forma usual de implementar un len-

guaje es definir un «lenguaje máquina» ideal y compilar el lenguaje para convertirlo en código para esta máquina. A continuación, un paso de traducción adicional convierte este código de máquina abstracta en código de máquina real.

Un ejemplo del modelo de capas es el modelo de referencia OSI de protocolos de red (Zimmermann, 1980), analizado en la Sección 11.5. Otro ejemplo que ha tenido cierta influencia fue propuesto por Buxton (Buxton, 1980), quien sugirió un modelo de tres capas para un Entorno de Soporte de Programación en Ada (APSE). La Figura 11.4 refleja la estructura de APSE y muestra cómo integrar un sistema de gestión de configuraciones usando esta aproximación de máquina abstracta.

El sistema de gestión de configuraciones gestiona las versiones de los objetos y proporciona facilidades de gestión de configuraciones generales, tal y como se explica en el Capítulo 29. Para soportar estas facilidades de gestión de configuraciones, se usa un sistema de gestión de objetos que proporciona almacenamiento de información y servicios de gestión para los elementos de configuración u objetos. El sistema se construye sobre un sistema de base de datos para proporcionar el almacenamiento básico de datos y servicios tales como gestión de transacciones, recuperación de actualizaciones y control de acceso. La gestión de la base de datos usa las facilidades del sistema operativo subyacente y almacenes de ficheros en su implementación. Pueden verse otros ejemplos de modelos arquitectónicos por capas en el Capítulo 13.

La aproximación por capas soporta el desarrollo incremental de sistemas. A medida que se desarrolla una capa, algunos de los servicios proporcionados por esa capa pueden estar disponibles para los usuarios. Esta arquitectura también soporta bien los cambios y es portable. En la medida en la que su interfaz permanezca sin cambios, una capa puede reemplazarse por otra capa equivalente. Además, cuando las interfaces de la capa cambian o se añaden nuevas facilidades a una capa, solamente se ve afectada la capa adyacente. Debido a que los sistemas por capas localizan las dependencias de la máquina en las capas más internas, es mucho más fácil proporcionar implementaciones multiplataforma de las aplicaciones de un sistema. Únicamente las capas más internas dependientes de la máquina necesitan ser reimplementadas para tener en cuenta las facilidades de un sistema operativo o base de datos diferente.

La desventaja de la aproximación por capas es que la estructuración de los sistemas puede resultar difícil. Las capas internas pueden proporcionar facilidades básicas, tales como ges-

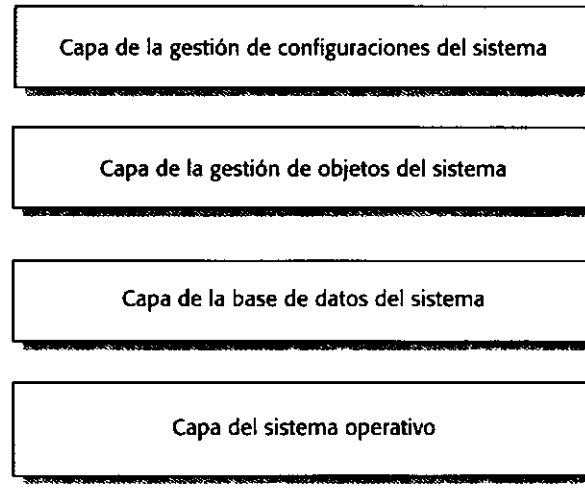


Figura 11.4 Modelo de capas de un sistema de gestión de versiones.

tión de ficheros, que son requeridas por todos los niveles. Los servicios requeridos por un usuario del nivel superior pueden, por lo tanto, tener que «atravesar» las capas adyacentes para tener acceso a los servicios proporcionados por los niveles inferiores. Esto trastoca el modelo, ya que implica que la capa más externa del sistema no solamente depende de su predecesora inmediata.

El rendimiento puede también ser un problema debido a que algunas veces se requieren múltiples niveles de interpretación de comandos. Si hay muchas capas, un servicio solicitado desde la capa superior puede tener que ser interpretado varias veces en diferentes capas antes de ser procesado. Para evitar estos problemas, las aplicaciones tienen que comunicarse directamente con las capas interiores en lugar de usar los servicios proporcionados por las capas adyacentes.

11.3 Estilos de descomposición modular

Después de que se haya elegido la organización del sistema en su totalidad, es necesario decidir la aproximación a usar para descomponer los subsistemas en módulos. No existe una distinción rígida entre la organización del sistema y la descomposición modular. Los estilos vistos en la Sección 11.2 podrían aplicarse a este nivel. Sin embargo, los componentes de los módulos son normalmente más pequeños que en los subsistemas, lo cual permite usar estilos alternativos de descomposición.

No hay una distinción clara entre subsistemas y módulos, pero resulta útil pensar sobre ellos de la siguiente forma:

1. Un subsistema es un sistema en sí mismo, cuyo funcionamiento no depende de los servicios proporcionados por otros subsistemas. Los subsistemas se componen de módulos y tienen interfaces definidas, las cuales se usan para comunicarse con otros subsistemas.
2. Un módulo es normalmente un componente de un subsistema que proporciona uno o más servicios a otros módulos. A su vez éste usa los servicios proporcionados por otros módulos. Esto no se suele considerar como un sistema independiente. Los módulos se componen normalmente de varios componentes del sistema más simples.

Hay dos estrategias principales que se pueden usar cuando se descomponga un subsistema en módulos:

1. *Descomposición orientada a objetos*, en la que se descompone un sistema en un conjunto de objetos que se comunican.
2. *Descomposición orientada a flujos de funciones*, en la que se descompone un sistema en módulos funcionales que aceptan datos y los transforman en datos de salida.

En la aproximación orientada a objetos, los módulos son objetos con estado privado y operaciones definidas sobre ese estado. En el modelo de flujos de funciones, los módulos son transformaciones funcionales. En ambos casos, los módulos pueden implementarse como componentes secuenciales o como procesos.

Debería evitarse tomar decisiones prematuras acerca de la concurrencia en un sistema. La ventaja de evitar el diseño concurrente de un sistema es que los programas secuenciales son más fáciles de diseñar, implementar, verificar y probar que los sistemas en paralelo. Las dependencias temporales entre los procesos son difíciles de formalizar, controlar y verificar. Es

mejor descomponer los sistemas en módulos, y entonces decidir durante la implementación si éstos necesitan ejecutarse secuencialmente o en paralelo.

11.3.1 Descomposición orientada a objetos

Un modelo arquitectónico orientado a objetos estructura el sistema en un conjunto de objetos débilmente acoplados y con interfaces bien definidas. Los objetos realizan llamadas a los servicios ofrecidos por otros objetos. Ya introdujimos los modelos de objetos en el Capítulo 8, y en el Capítulo 14 veremos con más detalle el diseño orientado a objetos.

La Figura 11.5 es un ejemplo de un modelo arquitectónico orientado a objetos de un sistema de procesamiento de facturas. Este sistema puede emitir facturas a los clientes, recibir pagos, emitir recibos para estos pagos y reclamaciones para las facturas no pagadas. se utiliza la notación UML introducida en el Capítulo 8 en donde las clases de objetos tienen nombres y un conjunto de atributos asociados. Las operaciones, si las hay, se definen en la parte inferior del rectángulo que representa al objeto. Las flechas discontinuas indican que un objeto usa los atributos o servicios proporcionados por otro objeto.

Una descomposición orientada a objetos está relacionada con las clases de objetos, sus atributos y sus operaciones. Cuando se implementa, los objetos se crean a partir de estas clases y se usan algunos modelos de control para coordinar las operaciones de los objetos. En este ejemplo particular, la clase **Factura** tiene varias operaciones asociadas que implementan la funcionalidad del sistema. Esta clase hace uso de otras clases que representan a los clientes, a los pagos y a los recibos.

Las ventajas de la aproximación orientada a objetos son bien conocidas. Debido a que los objetos están débilmente acoplados, la implementación de los objetos puede modificarse sin afectar a otros objetos. Los objetos son a menudo representaciones de entidades del mundo real por lo que la estructura del sistema es fácilmente comprensible. Debido a que las entidades del mundo real se usan en sistemas diferentes, los objetos pueden reutilizarse. Se han desarrollado lenguajes de programación orientados a objetos que proporcionan implementaciones directas de componentes arquitectónicos.

Sin embargo, la aproximación orientada a objetos tiene desventajas. Para utilizar los servicios, los objetos deben referenciar de forma explícita el nombre y la interfaz de otros objetos. Si se requiere un cambio de interfaz para satisfacer los cambios del sistema propuestos, se debe evaluar el efecto de ese cambio sobre todos los usuarios de los objetos cambiados. Si

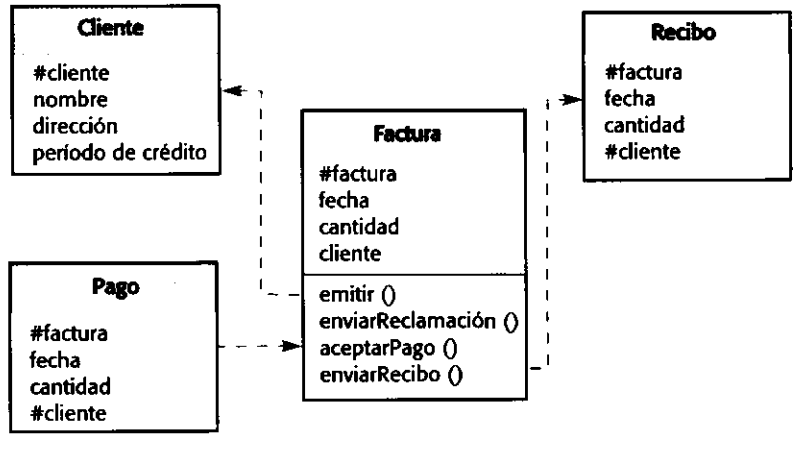


Figura 11.5 Un modelo de objetos de un sistema de procesamiento de facturas.

bien los objetos pueden corresponderse con entidades del mundo real a pequeña escala, algunas veces es difícil representar como objetos entidades más complejas.

11.3.2 Descomposición orientada a flujos de funciones

En una descomposición orientada a flujos de funciones o modelo de flujo de datos, las transformaciones funcionales procesan sus entradas y producen salidas. Los datos fluyen de una función a otra y se transforman a medida que se mueven a través de la secuencia de funciones. Cada paso de procesamiento se implementa como una transformación. Los datos de entrada fluyen a través de estas transformaciones hasta que se convierten en datos de salida. Las transformaciones se pueden ejecutar secuencialmente o en paralelo. Los datos pueden ser procesados por cada transformación elemento a elemento o en un único lote.

Cuando las transformaciones se representan como procesos separados, algunas veces este modelo se denomina modelo de tubería y filtro, siguiendo la terminología usada en el sistema Unix. El sistema Unix proporciona tuberías que actúan como conductoras de datos y un conjunto de comandos que son las transformaciones funcionales. Los sistemas que siguen este modelo pueden implementarse combinando comandos Unix que usan tuberías y las facilidades de control del intérprete de comandos de Unix. El término *filtro* se usa debido a que una transformación «filtra» los datos que puede procesar desde su flujo de datos de entrada.

Se han usado variantes de este modelo de flujo desde que las computadoras fueron utilizadas por primera vez para el procesamiento automático de datos. Cuando las transformaciones son secuenciales con datos procesados por lotes, este modelo arquitectónico es un modelo secuencial por lotes. Tal y como se indica en el Capítulo 13, ésta es una arquitectura común para sistemas de procesamiento de datos tales como sistemas de facturación. Los sistemas de procesamiento de datos normalmente generan muchos informes de salida que se derivan a partir de cálculos simples sobre un gran número de registros de entrada.

Un ejemplo de este tipo de arquitectura de sistemas se muestra en la Figura 11.6. Una organización ha emitido facturas a sus clientes. Una vez por semana, los pagos realizados se comparan con las facturas. Para esas facturas pagadas se emite un recibo. Para las facturas que no han sido pagadas dentro del plazo de pago se emite una reclamación.

Éste es un modelo solamente de una parte del sistema de procesamiento de facturas. Se podrían usar transformaciones alternativas para la emisión de facturas. Note la diferencia entre este modelo y su equivalente orientado a objetos comentado en la sección anterior. El modelo de objetos es más abstracto en tanto que no incluye información sobre la secuencia de operaciones.

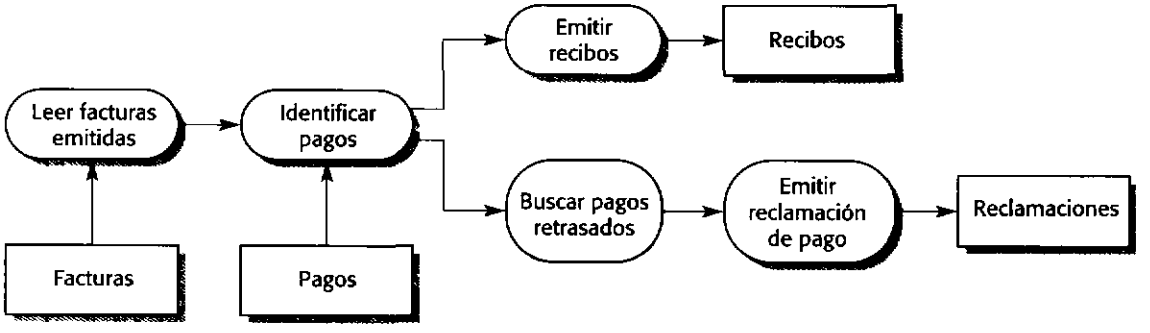


Figura 11.6 Un modelo de flujo de funciones de un sistema de procesamiento de facturas.

Las ventajas de esta arquitectura son las siguientes:

1. Permite la reutilización de transformaciones.
2. Es intuitiva puesto que muchas personas piensan en su trabajo en términos de procesamiento de entradas y salidas.
3. Generalmente se puede hacer evolucionar de forma directa el sistema añadiendo nuevas transformaciones.
4. Es sencilla de implementar ya sea como un sistema concurrente o como uno secuencial.

El principal problema con este estilo es que tiene que haber un formato común para transferir los datos de forma que puedan ser reconocidos por todas las transformaciones. Cada transformación debe estar acorde con las transformaciones con las que se comunica sobre el formato de los datos a procesar o bien se debe imponer un formato estándar para todos los datos comunicados. Esto último es la única aproximación factible cuando las transformaciones son independientes y reutilizables. En Unix, el formato estándar es simplemente una secuencia de caracteres. Cada transformación debe analizar su entrada y proporcionar la salida en el formato acordado. Esto incrementa la sobrecarga del sistema y puede significar que sea imposible integrar transformaciones que utilicen formatos incompatibles de datos.

Los sistemas interactivos son difíciles de describir usando el modelo de flujo de funciones debido a la necesidad de un flujo de datos a procesar. Mientras que un modelo textual sencillo de entradas y salidas puede modelarse de esta forma, las interfaces gráficas de usuario tienen formatos de entrada/salida más complejos y controles, los cuales se basan en eventos tales como pulsaciones del ratón o selecciones de menús. Es difícil traducir esto a una forma compatible con el modelo de flujo de funciones.

11.4 Estilos de control

Los modelos para estructurar un sistema están relacionados con la forma en que un sistema se descompone en subsistemas. Para trabajar como un sistema, los subsistemas deben ser controlados para que sus servicios se entreguen en el lugar correcto en el momento preciso. Los modelos estructurales no incluyen (y no deberían hacerlo) información de control. En su lugar, el arquitecto debería organizar los subsistemas de acuerdo con algún modelo de control que complemente el modelo de estructura usado. Los modelos de control a nivel arquitectónico están relacionados con el flujo de control entre subsistemas.

Hay dos estilos de control genéricos que se usan en sistemas software:

1. *Control centralizado.* Un subsistema tiene toda la responsabilidad para controlar e iniciar y detener a otros subsistemas. También puede devolver el control a otro subsistema, pero esperará que le sea devuelta la responsabilidad del control.
2. *Control basado en eventos.* En lugar de que la información de control esté embebida en un subsistema, cada subsistema puede responder a eventos generados externamente. Estos eventos podrían provenir de otros subsistemas o del entorno del sistema.

Los estilos de control se usan conjuntamente con estilos estructurales. Todos los estilos estructurales analizados se pueden llevar a cabo utilizando control centralizado o control basado en eventos.

11.4.1 Control centralizado

En un modelo de control centralizado, un subsistema se diseña como el controlador del sistema y tiene la responsabilidad de gestionar la ejecución de otros subsistemas. Los modelos de control centralizado se dividen en dos clases, según que los subsistemas controlados se ejecuten secuencialmente o en paralelo.

1. *El modelo de llamada-retorno.* Es el modelo usual de subrutina descendente en donde el control comienza al inicio de una jerarquía de subrutinas y, a través de las llamadas a subrutinas, el control pasa a niveles inferiores en el árbol de la jerarquía. El modelo de subrutinas solamente es aplicable a sistemas secuenciales.
2. *El modelo del gestor.* Éste es aplicable a sistemas concurrentes. Un componente del sistema se diseña como un gestor del sistema y controla el inicio, parada y coordinación del resto de los procesos del sistema. Un proceso es un subsistema o módulo que puede ejecutarse en paralelo con otros procesos. Una variante de este modelo también puede aplicarse a sistemas secuenciales en los que la rutina de gestión llama a subsistemas particulares dependiendo de los valores de algunas variables de estado. Esto normalmente se implementa como una sentencia *case*.

El modelo de llamada-retorno se ilustra en la Figura 11.7. El programa principal puede llamar a las Rutinas 1, 2 y 3; la Rutina 1 puede llamar a las Rutinas 1.1 o 1.2; la Rutina 3 puede llamar a las rutinas 3.1 o 3.2; y así sucesivamente. Éste es un modelo que muestra la dinámica del programa. *No* es un modelo estructural; no es necesario que la Rutina 1.1, por ejemplo, sea parte de la Rutina 1.

Este modelo familiar está embebido en lenguajes de programación tales como C, Ada y Pascal. Las subrutinas en un lenguaje de programación que son llamadas por otras subrutinas son naturalmente funcionales. Sin embargo, en muchos sistemas orientados a objetos, las operaciones sobre los objetos (métodos) se implementan como procedimientos o funciones. Por ejemplo, cuando un objeto Java solicita un servicio de otro objeto, lo hace llamando a un método asociado a dicho objeto.

La naturaleza rígida y restrictiva de este modelo es tanto una ventaja como un inconveniente. Es una ventaja debido a que es relativamente simple analizar flujos de control y conocer cómo responderá el sistema a cierto tipo de entradas. Es un inconveniente debido a que las excepciones a las operaciones normales son tediosas de manejar.

La Figura 11.8 ilustra un modelo de gestión de control centralizado para un sistema concurrente. Este modelo se usa a menudo en sistemas de tiempo real «blandos», los cuales no tienen restricciones de tiempo muy estrictas. El controlador central gestiona la ejecución de

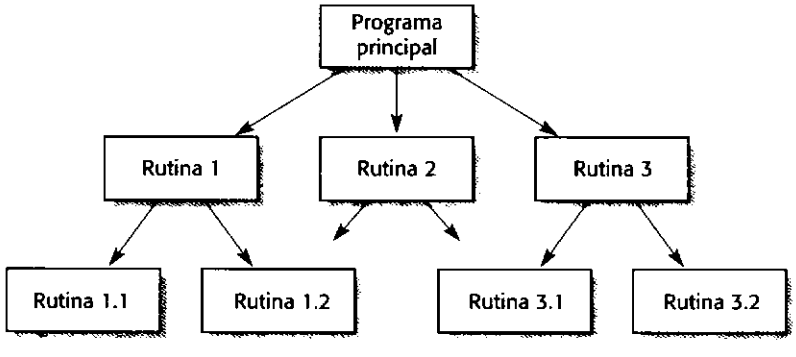


Figura 11.7 El modelo de control de llamada-retorno.

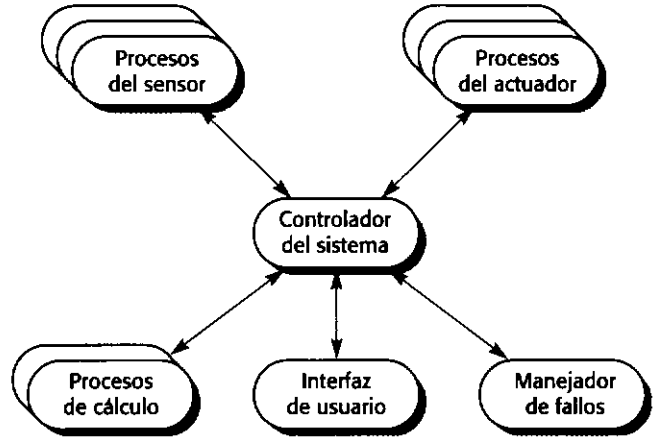


Figura 11.8 Un modelo de control centralizado para un sistema de tiempo real.

un conjunto de procesos asociados con sensores y actuadores. La construcción del sistema de monitorización explicada en el Capítulo 15 usa este modelo de control.

El proceso controlador del sistema decide cuándo deberían comenzar o terminar los procesos dependiendo de las variables de estado del sistema. El sistema comprueba si otros procesos han producido información para ser procesada o para enviarles información para su procesamiento. El controlador por lo general realiza ciclos continuamente, consultando los sensores y otros procesos para detectar eventos o cambios de estado. Por esta razón, este modelo se llama algunas veces modelo de ciclo de eventos.

11.4.2 Sistemas dirigidos por eventos

En los modelos de control centralizados, las decisiones de control se determinan normalmente por los valores de algunas variables de estado del sistema. Por el contrario, los modelos de control dirigidos por eventos se rigen por eventos generados externamente. El término *evento* en este contexto no sólo significa una señal binaria. Puede ser una señal dentro de un rango de valores o una entrada de un comando desde un menú. La diferencia entre un evento y una entrada simple es que la aparición del evento está fuera del control del proceso que maneja ese evento.

Hay muchos tipos de sistemas dirigidos por eventos. Éstos incluyen editores, en donde los eventos de la interfaz de usuario provocan la ejecución de comandos; sistemas de producción basados en reglas, como los usados en AI, en donde una condición que se convierte en verdadera provoca que se dispare una acción, y los objetos activos, en los que cambiar un valor de un atributo del objeto dispara algunas acciones. Garlan y otros (Garlan *et al.*, 1992) estudian estos diferentes tipos de sistemas.

En esta sección, se analizan dos modelos de control dirigidos por eventos:

1. *Modelos de transmisión (broadcast).* En estos modelos, un evento se transmite a todos los subsistemas. Cualquier subsistema que haya sido programado para manejar ese evento puede responder a él.
2. *Modelos dirigidos por interrupciones.* Éstos se usan exclusivamente en sistemas de tiempo real, en donde las interrupciones externas son detectadas por un manejador de interrupciones. A continuación, éstas se envían a algún otro componente para su procesamiento.

Los modelos de transmisión son efectivos para integrar subsistemas distribuidos en diferentes computadoras de una red. Los modelos dirigidos por interrupciones se usan en sistemas de tiempo real con requerimientos temporales rígidos.

En un modelo de transmisión (Figura 11.9), los subsistemas registran un interés en eventos específicos. Cuando estos eventos ocurren, el control se transfiere al subsistema que puede manejar el evento. La diferencia entre este modelo y el modelo centralizado mostrado en la Figura 11.8 es que la política de control no está embebida en el manejador de eventos y mensajes. Los subsistemas deciden qué eventos requieren y el manejador de eventos y mensajes asegura que estos eventos sean enviados a dichos subsistemas.

Todos los eventos podrían enviarse a todos los subsistemas, pero esto supondría una gran sobrecarga de procesamiento. A menudo, el manejador de eventos y de mensajes mantiene un registro de los subsistemas y de los eventos que a éstos les interesan. Los subsistemas generan eventos que indican, quizás, que algún dato está disponible para su procesamiento. El manejador de eventos detecta los eventos, consulta el registro de eventos y envía el evento a aquellos subsistemas que han declarado un interés por él.

En sistemas más simples, tales como sistemas basados en PCs dirigidos por eventos de interfaz de usuario, hay subsistemas explícitos que actúan como oyentes de eventos que están a la espera de eventos provenientes del ratón, el teclado, y otros, y los traducen en comandos más específicos.

El manejador de eventos también soporta normalmente comunicaciones punto a punto. Un subsistema puede enviar un mensaje de forma explícita a otro subsistema. Hay diversas variantes de este modelo, tales como el entorno de Field (Reiss, 1990) y el Softbench de Hewlett-Packard (Fromme y Walker, 1993). Ambos se han utilizado para controlar las interacciones de las herramientas en entornos de ingeniería del software. Los intermediarios de peticiones de objetos (ORBs), estudiados en el Capítulo 12, también soportan este modelo de control para comunicaciones de objetos distribuidos.

La ventaja de esta aproximación de transmisión (*broadcast*) es que la evolución es relativamente simple. Un nuevo subsistema puede integrarse para manejar clases particulares de eventos registrando sus eventos con el manejador de eventos. Cualquier subsistema puede activar otros subsistemas sin conocer su nombre o localización. Los subsistemas pueden implementarse sobre máquinas distribuidas. Esta distribución es transparente para el resto de los subsistemas.

La desventaja de este modelo es que los subsistemas no conocen si los eventos se manejarán ni cuándo serán manejados. Cuando un subsistema genera un evento no conoce qué subsistemas han registrado un interés en dicho evento. Es bastante probable que subsistemas diferentes se registren para los mismos eventos. Esto puede ocasionar conflictos cuando están disponibles los resultados del manejo del evento.

Los sistemas de tiempo real que requieren que los eventos generados externamente sean manejados muy rápidamente, deben ser conducidos por eventos. Por ejemplo, si un sistema

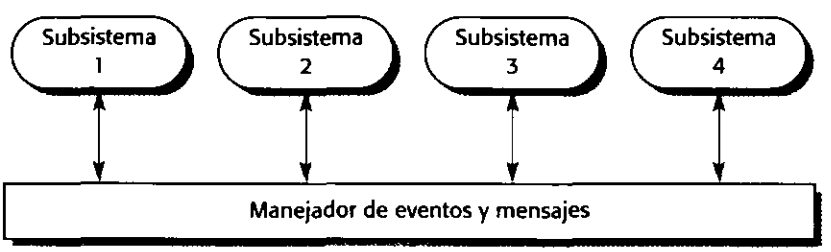


Figura 11.9
Un modelo de control basado en transmisión selectiva.

de tiempo real se usa para controlar la seguridad del sistema en un vehículo, debe detectar un posible choque y, quizás, inflar el airbag antes de que la cabeza del conductor impacte sobre el volante. Para proporcionar esta respuesta rápida a los eventos, se debe usar un control dirigido por interrupciones.

Un modelo de control dirigido por interrupciones se ilustra en la Figura 11.10. Hay varios tipos de interrupciones conocidos con un manejador definido para cada uno de ellos. Cada tipo de interrupción se asocia con una localización de memoria en donde se almacena la dirección del manejador. Cuando se recibe una interrupción de un tipo particular, un conmutador hardware hace que el control se transfiera inmediatamente a este manejador. Este manejador de interrupciones puede entonces iniciar o detener a otros procesos en respuesta al evento señalizado por la interrupción.

Este modelo se usa principalmente en sistemas de tiempo real en los que es necesaria una respuesta inmediata a algún evento. Puede ser combinada con el modelo de gestión centralizado. El gestor central maneja la ejecución normal del sistema con un control para emergencias basado en interrupciones.

La ventaja de esta aproximación es que permite implementar respuestas muy rápidas a los eventos. Sus desventajas son que es complejo de programar y difícil de validar. Puede ser imposible replicar patrones de ocurrencias de las interrupciones durante la prueba del sistema. Puede ser difícil cambiar los sistemas desarrollados usando este modelo si el número de interrupciones está limitado por el hardware. Una vez que se alcanza este límite, no puede manejarse ningún otro tipo de eventos. Se puede algunas veces obviar esta limitación haciendo corresponder varios tipos de eventos con una única interrupción. El manejador entonces conoce qué evento ha tenido lugar. Sin embargo, la correspondencia de interrupciones puede no ser práctica si se requiere una respuesta muy rápida a interrupciones individuales.

11.5 Arquitecturas de referencia

Los modelos arquitectónicos citados anteriormente son modelos generales: pueden aplicarse a muchas clases de aplicaciones. Al igual que los modelos generales, también pueden usarse los modelos arquitectónicos que son específicos para un dominio particular de aplicación. Si bien las instancias de estos sistemas difieren en los detalles, la estructura arquitectónica co-

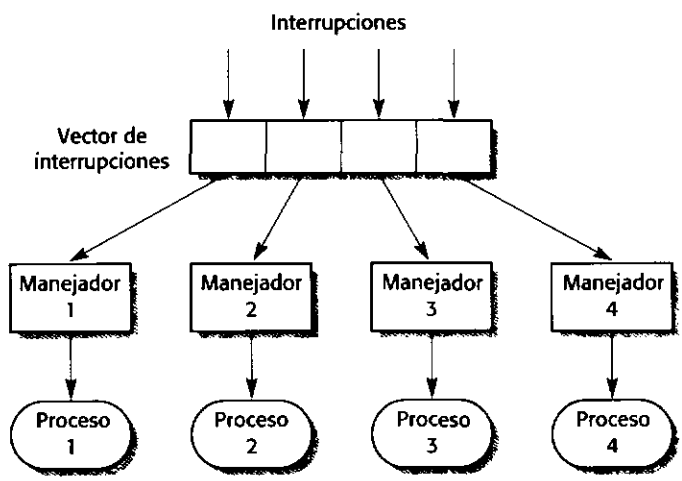


Figura 11.10 Un modelo de control conducido por interrupciones.

mún puede reutilizarse cuando se desarrollan nuevos sistemas. Estos modelos arquitectónicos se denominan *arquitecturas de dominio específico*.

Hay dos tipos de modelos arquitectónicos de dominio específico:

1. *Modelos genéricos*. Son abstracciones obtenidas a partir de varios sistemas reales. Encapsulan las características principales de estos sistemas. Por ejemplo, en sistemas de tiempo real, podría haber modelos arquitectónicos genéricos de diferentes tipos de sistemas tales como sistemas de recolección de datos o sistemas de monitorización. Se estudian varios modelos genéricos en el Capítulo 13, que incluye las arquitecturas de aplicaciones. Esta sección se centra en los modelos de referencia arquitectónicos.
2. *Modelos de referencia*. Son más abstractos y describen una clase más amplia de sistemas. Constituyen un modo de informar a los diseñadores sobre la estructura general de esta clase de sistemas. Los modelos de referencia normalmente se obtienen a partir de un estudio del dominio de la aplicación. Representan una arquitectura ideal que incluye todas las características que los sistemas podrían incorporar.

No hay, desde luego, una distinción rígida entre estos tipos de modelos. Los modelos genéricos también pueden servir como modelos de referencia. Hacemos aquí la distinción entre ellos debido a que los modelos genéricos pueden reutilizarse directamente en un diseño. Los modelos de referencia se usan normalmente para comunicar conceptos del dominio y comparar o evaluar posibles arquitecturas.

Las arquitecturas de referencia normalmente no se consideran como un camino para la implementación. En su lugar, su principal función es una forma de tratar arquitecturas específicas del dominio y de comparar sistemas diferentes en un dominio. Un modelo de referencia proporciona un vocabulario para realizar comparaciones. Dicho modelo actúa como una base, frente a la cual los sistemas pueden ser evaluados.

El modelo OSI es un modelo de siete capas para la interconexión de sistemas abiertos. El modelo se ilustra en la Figura 11.11. Las funciones exactas de las capas no son importantes aquí. En esencia, las capas inferiores están relacionadas con la interconexión física, las capas intermedias con la transferencia de los datos y las capas superiores con la transferencia de información de la aplicación semánticamente significativa como documentos estandarizados.

Los diseñadores del modelo OSI tuvieron el objetivo práctico de definir una implementación estándar para que los sistemas acordes con ella pudiesen comunicarse unos con otros.

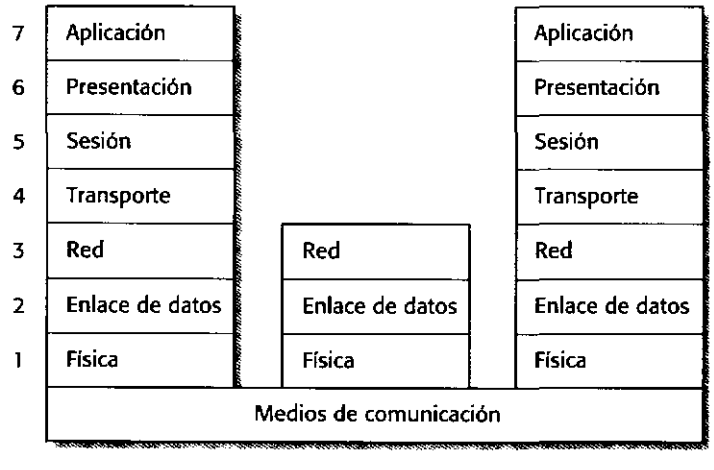


Figura 11.11 Arquitectura del modelo de referencia OSI.

Cada capa solamente debería depender de la capa inmediatamente inferior. A medida que se desarrollan nuevas tecnologías, una capa podría ser reimplementada de forma transparente sin afectar a los sistemas que usan el resto de las capas.

En la práctica, sin embargo, los problemas de rendimiento de la aproximación por capas para el modelado arquitectónico han comprometido este objetivo. Debido a las grandes diferencias entre las redes, una simple interconexión puede ser imposible. Si bien las características funcionales de cada capa están bien definidas, las características no funcionales no están definidas. Los desarrolladores del sistema tienen que implementar sus propias facilidades de más alto nivel y omitir algunas capas del modelo. De forma alternativa, tienen que diseñar características no estándares para mejorar el rendimiento del sistema.

Como consecuencia, el reemplazo de una capa en el modelo de forma transparente es realmente muy difícil. Sin embargo, esto no niega la utilidad del modelo ya que proporciona una base para la estructuración abstracta y la implementación sistemática de comunicaciones entre los sistemas.

Otro modelo de referencia propuesto es un modelo de referencia para entornos CASE (ECMA, 1991; Brown *et al.*, 1992) que identifica cinco conjuntos de servicios que un entorno CASE debería proporcionar. También debería proporcionar facilidades «plug-in» para herramientas CASE individuales que utilizan estos servicios. El modelo de referencia CASE se ilustra en la Figura 11.12. Los cinco niveles de servicio en el modelo de referencia CASE son:

1. *Servicios de repositorio de datos.* Proporcionan facilidades para el almacenamiento y gestión de los elementos de datos y sus relaciones.
2. *Servicios de integración de datos.* Proporcionan facilidades para gestionar grupos o el establecimiento de relaciones entre ellos. Estos servicios y los servicios de repositorio de datos son las bases de la integración de datos en el entorno.
3. *Servicios de gestión de tareas.* Proporcionan facilidades para la definición y establecimiento de normas de los modelos de proceso. Soportan integración de procesos.
4. *Servicios de mensajes.* Éstos proporcionan facilidades para comunicaciones herramienta-herramienta, entorno-herramienta y entorno-entorno. Soportan la integración del control.
5. *Servicios de interfaz de usuario.* Proporcionan facilidades para el desarrollo de interfaces de usuario. Soportan la integración de la presentación.

Este modelo de referencia nos dice lo que debería incluirse en cualquier entorno CASE particular, si bien es importante destacar que no se incluirá en los diseños arquitectónicos reales cada característica de una arquitectura de referencia. Esto significa que nosotros podemos

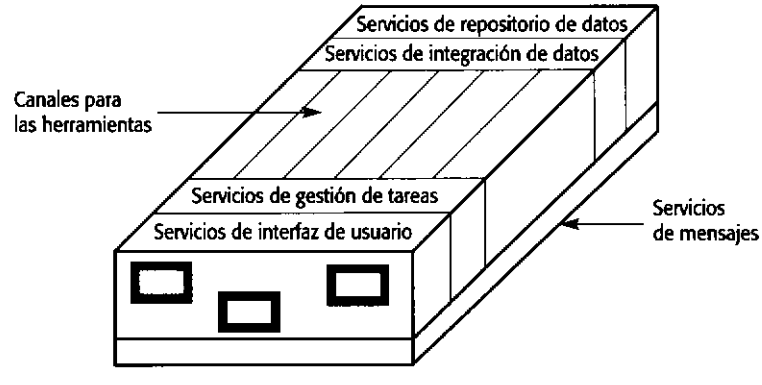


Figura 11.12
La arquitectura de referencia ECMA para entornos CASE.

plantear cuestiones sobre el diseño de un sistema del tipo «¿cómo se proporcionan los servicios del repositorio de datos?» y «¿el sistema proporciona gestión de tareas?».

Una vez más, el principal valor de estas arquitecturas de referencia es que sirven como una forma de clasificar y comparar entornos y herramientas CASE integradas. Además, también pueden usarse en entornos educativos para resaltar las características clave de estos entornos y considerarlos de una forma genérica.

PUNTOS CLAVE

- **La arquitectura del software es el marco fundamental para estructurar el sistema. Las propiedades de un sistema tales como rendimiento, seguridad y disponibilidad están influenciadas por la arquitectura utilizada.**
- **Las decisiones de diseño arquitectónico incluyen decisiones sobre el tipo de aplicación, la distribución del sistema, los estilos arquitectónicos a utilizar y las formas en las que la arquitectura debería documentarse y evaluarse.**
- **Diferentes modelos arquitectónicos tales como un modelo estructural, un modelo de control y un modelo de descomposición pueden ser desarrollados durante el proceso de diseño arquitectónico.**
- **Los modelos organizacionales de un sistema comprenden los modelos de repositorio, los modelos cliente-servidor y los modelos de máquina abstracta. Los modelos de repositorio comparten datos a través de un almacén común. Los modelos cliente-servidor normalmente distribuyen los datos. Los modelos de máquina abstracta se organizan en capas, implementando cada capa utilizando las facilidades proporcionadas por la capa adyacente.**
- **Los estilos de descomposición comprenden la descomposición orientada a funciones y la orientada a objetos. Los modelos orientados a flujo son funcionales y los modelos de objetos se basan en entidades pobremente acopladas que mantienen su propio estado y operaciones.**
- **Los estilos de control incluyen control centralizado y control basado en eventos. En los modelos centralizados de control, las decisiones de control se toman dependiendo del estado del sistema; en los modelos de eventos, los eventos externos controlan el sistema.**
- **Las arquitecturas de referencia pueden usarse como una forma de estudiar arquitecturas específicas del dominio y evaluar y comparar diseños arquitectónicos.**

LECTURAS ADICIONALES

Software Architecture in Practice, 2nd ed. Es un estudio práctico de arquitecturas software sin exagerar en su presentación y que proporciona una clara exposición razonada para las empresas de por qué las arquitecturas son importantes. (L. Bass *et al.*, 2003, Addison-Wesley.)

Design and Use of Software Architectures. Si bien este libro se centra en arquitecturas de líneas de productos, los primeros capítulos son una introducción excelente a las cuestiones generales en el diseño de la arquitectura del software. (J. Bosch, 2000, Addison-Wesley.)

Software Architecture: Perspectives on an Emerging Discipline. Éste fue el primer libro sobre arquitecturas software y tiene un buen estudio sobre diferentes estilos arquitectónicos. (M. Shaw y D. Garlan, 1996, Prentice-Hall.)

EJERCICIOS

11.1 Explique por qué puede ser necesario diseñar la arquitectura del sistema antes de redactar las especificaciones.

11.2 Explique por qué podrían tener lugar conflictos de diseño al diseñar una arquitectura en la que los requerimientos de disponibilidad y seguridad son los requerimientos no funcionales más importantes.

11.3 Construya una tabla que muestre las ventajas e inconvenientes de los modelos estructurales analizados en este capítulo.

11.4 Sugiera, justificando sus respuesta, un modelo estructural adecuado para los siguientes sistemas:

- Un sistema de venta automática de billetes usados por los pasajeros en una estación de trenes.
- Un sistema de videoconferencia controlada por computadora que permita que el vídeo, audio y datos de la computadora estén disponibles para varios participantes al mismo tiempo.
- Un robot limpiador de suelos que tenga como objetivo limpiar espacios relativamente vacíos tales como pasillos. El limpiador debe ser capaz de detectar paredes y otros obstáculos.

11.5 Diseñe una arquitectura para los sistemas anteriores basada en el modelo que usted ha elegido. Haga suposiciones razonables sobre los requerimientos del sistema.

11.6 Los sistemas de tiempo real normalmente utilizan modelos de control dirigidos por eventos. ¿En qué circunstancias podría recomendar el uso de un modelo de control de llamada-retorno para un sistema de tiempo real?

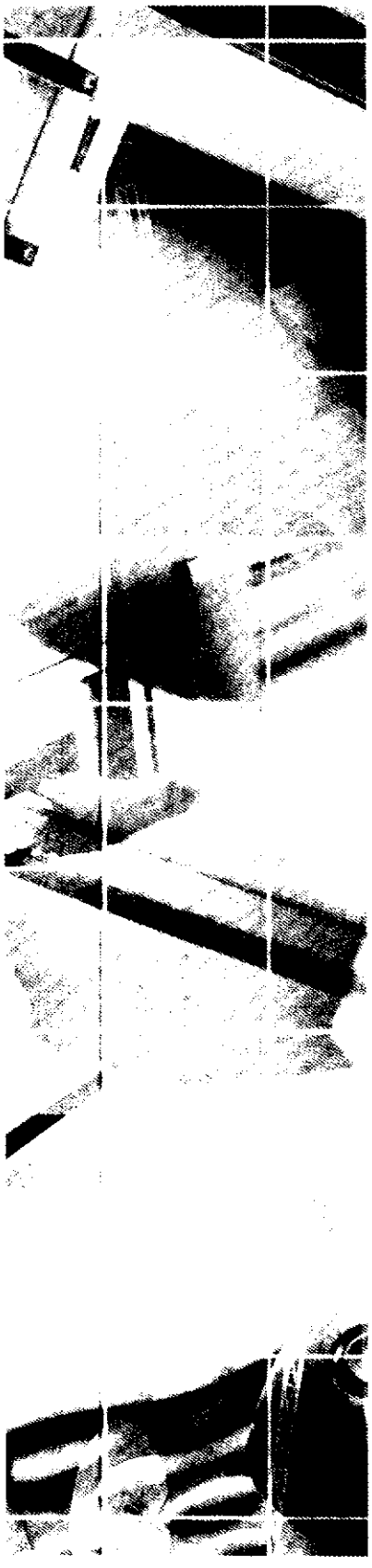
11.7 Sugiera, justificando su respuesta, un modelo de control adecuado para los siguientes sistemas:

- Un sistema de procesamiento por lotes que tenga como entrada la información sobre las horas trabajadas y tarifas de pago e imprima información sobre hojas de salarios y la información bancaria de la transferencia de éstos.
- Un conjunto de herramientas software que son producidas por diferentes vendedores, pero que tienen que trabajar conjuntamente.
- Un controlador de televisión que responde a las señales de una unidad de control remoto.

11.8 Comente las ventajas e inconvenientes relacionados con la capacidad de distribución del modelo de flujo de datos y el modelo de objetos. Suponga que se requieren tanto versiones distribuidas como versiones con una única máquina.

11.9 Se le han proporcionado dos conjuntos de herramientas CASE integradas y se le ha solicitado que las compare. Explique cómo podría usar un modelo de referencia para herramientas CASE (Brown *et al.*, 1992) para hacer esta comparación.

11.10 ¿Debería existir una profesión de «arquitecto software» cuyo cometido fuese trabajar de forma independiente con un cliente para diseñar una arquitectura de un sistema software? El sistema entonces debería ser implementado por alguna compañía de software. ¿Cuáles podrían ser las dificultades del establecimiento de una profesión como ésta?



12 Arquitecturas de sistemas distribuidos

Objetivos

El objetivo de este capítulo es estudiar modelos de arquitectura del software para sistemas distribuidos. Cuando haya leído este capítulo:

- conocerá las ventajas y desventajas de las arquitecturas de sistemas distribuidos;
- comprenderá los dos modelos principales de arquitecturas de sistemas distribuidos, denominados sistemas cliente-servidor y sistemas de objetos distribuidos;
- comprenderá el concepto de un objeto intermediario de peticiones y los principios que subyacen a los estándares CORBA;
- se habrá introducido en las arquitecturas de igual a igual (peer-to-peer) y arquitecturas orientadas a servicios.

Contenidos

Prácticamente todos los grandes sistemas informáticos son en la actualidad sistemas distribuidos. Un sistema distribuido es un sistema en el que el procesamiento de información se distribuye sobre varias computadoras en vez de estar confinado en una única máquina. Obviamente, la ingeniería de sistemas distribuidos tiene mucho en común con la ingeniería de cualquier otro software, pero existen cuestiones específicas que deben tenerse en cuenta cuando se diseña este tipo de sistemas. Ya se han presentado algunas de estas cuestiones en la introducción a las arquitecturas cliente-servidor en el Capítulo 11 y se explican aquí con mayor detalle.

Coulouris y otros (Coulouris *et al.*, 2001) estudian las características importantes de los sistemas distribuidos. Identifican las siguientes ventajas del uso de una aproximación distribuida para el desarrollo de sistemas:

1. *Compartición de recursos.* Un sistema distribuido permite compartir recursos hardware y software —como discos, impresoras, ficheros y compiladores— que se asocian con computadoras de una red.
2. *Apertura.* Los sistemas distribuidos son normalmente sistemas abiertos, lo que significa que se diseñan sobre protocolos estándar que permiten combinar equipamiento y software de diferentes vendedores.
3. *Concurrencia.* En un sistema distribuido, varios procesos pueden operar al mismo tiempo sobre diferentes computadoras de la red. Estos procesos pueden (aunque no necesariamente) comunicarse con otros durante su funcionamiento normal.
4. *Escalabilidad.* Al menos en principio, los sistemas distribuidos son escalables en tanto que la capacidad del sistema puede incrementarse añadiendo nuevos recursos para cubrir nuevas demandas sobre el sistema. En la práctica, la red que une las computadoras individuales del sistema puede limitar la escalabilidad del sistema. Si se añaden muchas computadoras nuevas, entonces la capacidad de la red puede resultar inadecuada.
5. *Tolerancia a defectos.* La disponibilidad de varias computadoras y el potencial para reproducir información significa que los sistemas distribuidos pueden ser tolerantes a algunos fallos de funcionamiento del hardware y del software (véase el Capítulo 20). En la mayoría de los sistemas distribuidos, se puede proporcionar un servicio degradado cuando ocurren fallos de funcionamiento; una completa pérdida de servicio sólo ocurre cuando existe un fallo de funcionamiento en la red.

Para sistemas organizacionales a gran escala, estas ventajas significan que los sistemas distribuidos han reemplazado ampliamente a los sistemas heredados centralizados que fueron desarrollados en los años 80 y 90. Sin embargo, comparados con sistemas que se ejecutan sobre un único procesador o un cluster de procesadores, los sistemas distribuidos tienen varias desventajas:

1. *Complejidad.* Los sistemas distribuidos son más complejos que los sistemas centralizados. Esto hace más difícil comprender sus propiedades emergentes y probar estos sistemas. Por ejemplo, en vez de que el rendimiento del sistema dependa de la velocidad de ejecución de un procesador, depende del ancho de banda y de la velocidad de los procesadores de la red. Mover los recursos de una parte del sistema a otra puede afectar de forma radical al rendimiento del sistema.
2. *Seguridad.* Puede accederse al sistema desde varias computadoraes diferentes, y el tráfico en la red puede estar sujeto a escuchas indeseadas. Esto hace más difícil el asegurar que la integridad de los datos en el sistema se mantenga y que los servicios del sistema no se degraden por ataques de denegación de servicio.

3. *Manejabilidad.* Las computadoras en un sistema pueden ser de diferentes tipos y pueden ejecutar versiones diferentes de sistemas operativos. Los defectos en una máquina pueden propagarse a otras máquinas con consecuencias inesperadas. Esto significa que se requiere más esfuerzo para gestionar y mantener el funcionamiento del sistema.
4. *Impredecibilidad.* Como todos los usuarios de la WWW saben, los sistemas distribuidos tienen una respuesta impredecible. La respuesta depende de la carga total en el sistema, de su organización y de la carga de la red. Como todos ellos pueden cambiar con mucha rapidez, el tiempo requerido para responder a una petición de usuario puede variar drásticamente de una petición a otra.

El reto para el diseño es diseñar el software y hardware para proporcionar características deseables a los sistemas distribuidos y, al mismo tiempo, minimizar los problemas inherentes a estos sistemas. Para hacer eso, se necesita comprender las ventajas y desventajas de las diferentes arquitecturas de sistemas distribuidos. Aquí se tratan dos tipos genéricos de arquitecturas de sistemas distribuidos:

1. *Arquitecturas cliente-servidor.* En esta aproximación, el sistema puede ser visto como un conjunto de servicios que se proporcionan a los clientes que hacen uso de dichos servicios. Los servidores y los clientes se tratan de forma diferente en estos sistemas.
2. *Arquitecturas de objetos distribuidos.* En este caso, no hay distinción entre servidores y clientes, y el sistema puede ser visto como un conjunto de objetos que interaccionan cuya localización es irrelevante. No hay distinción entre un proveedor de servicios y el usuario de estos servicios.

Ambas arquitecturas se usan ampliamente en la industria, pero la distribución de las aplicaciones generalmente tiene lugar dentro de una única organización. La distribución soportada es, por lo tanto, intraorganizacional. Aquí también planteamos dos tipos más de arquitecturas distribuidas que son más adecuadas para la distribución interorganizacional: arquitectura de sistemas peer-to-peer (p2p) y arquitecturas orientadas a servicios. Los sistemas peer-to-peer han sido usados principalmente para sistemas personales, pero están comenzando a usarse para aplicaciones de empresa. En el momento de escribir este libro, se estaban introduciendo los sistemas orientados a servicios, pero la aproximación orientada a servicios probablemente se convertirá en un modelo de distribución significativo en 2005.

Los componentes en un sistema distribuido pueden implementarse en diferentes lenguajes de programación y pueden ejecutarse en tipos de procesadores completamente diferentes. Los modelos de datos, la representación de la información y los protocolos de comunicación pueden ser todos diferentes. Un sistema distribuido, por lo tanto, requiere software que pueda gestionar estas partes distintas, y asegurar que dichas partes se puedan comunicar e intercambiar datos. El término *middleware* se usa para hacer referencia a ese software; se sitúa en medio de los diferentes componentes distribuidos del sistema.

Bernstein (Bernstein, 1996) resume los tipos de middleware disponibles para soportar computación distribuida. El middleware es un software de propósito general que normalmente se compra como un componente comercial más que escribirse especialmente por los desarrolladores de la aplicación. Ejemplos de middleware son software para gestionar comunicaciones con bases de datos, administradores de transacciones, convertidores de datos y controladores de comunicación. Más adelante en este capítulo se describen los denominados intermediarios de peticiones de objetos (*object request broker*), que constituyen una clase importante de middleware para sistemas distribuidos.

Los sistemas distribuidos se desarrollan normalmente utilizando una aproximación orientada a objetos. Estos sistemas están formados por partes independientes pobremente integradas, cada una de las cuales puede interaccionar directamente con los usuarios o con otras partes del sistema. Algunas partes del sistema pueden tener que responder a eventos independientes. Los objetos software reflejan estas características; por lo tanto, son abstracciones naturales para los componentes de sistemas distribuidos.

12.1 Arquitecturas multiprocesador

El modelo más simple de un sistema distribuido es un sistema multiprocesador en el que el sistema software está formado por varios procesos que pueden (aunque no necesariamente) ejecutarse sobre procesadores diferentes. Este modelo es común en sistemas grandes de tiempo real. Tal y como se expone en el Capítulo 15, estos sistemas recogen información, toman decisiones usando esta información y envían señales a los actuadores que modifican el entorno del sistema.

Lógicamente, los procesos relacionados con la recopilación de información, toma de decisiones y control de actuadores podrían ejecutarse todos ellos sobre un único procesador bajo el control de un planificador (*scheduler*). El uso de múltiples procesadores mejora el rendimiento y adaptabilidad del sistema. La distribución de procesos entre los procesadores puede ser predeterminada (esto es común en sistemas críticos) o puede estar bajo el control de un despachador (*dispatcher*) que decide qué procesos se asignan a cada procesador.

Un ejemplo de este tipo de sistemas se muestra en la Figura 12.1. Éste es un modelo simplificado de sistema de control de tráfico. Un conjunto de sensores distribuidos recogen información sobre el flujo de tráfico y la procesan localmente antes de enviarla a una sala de control. Los operadores toman decisiones usando esta información y dan instrucciones a un proceso de control de diversas luces de tráfico. En este ejemplo, hay varios procesos lógicos para gestionar los sensores, la sala de control y los semáforos. Estos procesos lógicos pueden ser procesos individuales o un grupo de procesos. En este ejemplo, se ejecutan sobre procesadores diferentes.

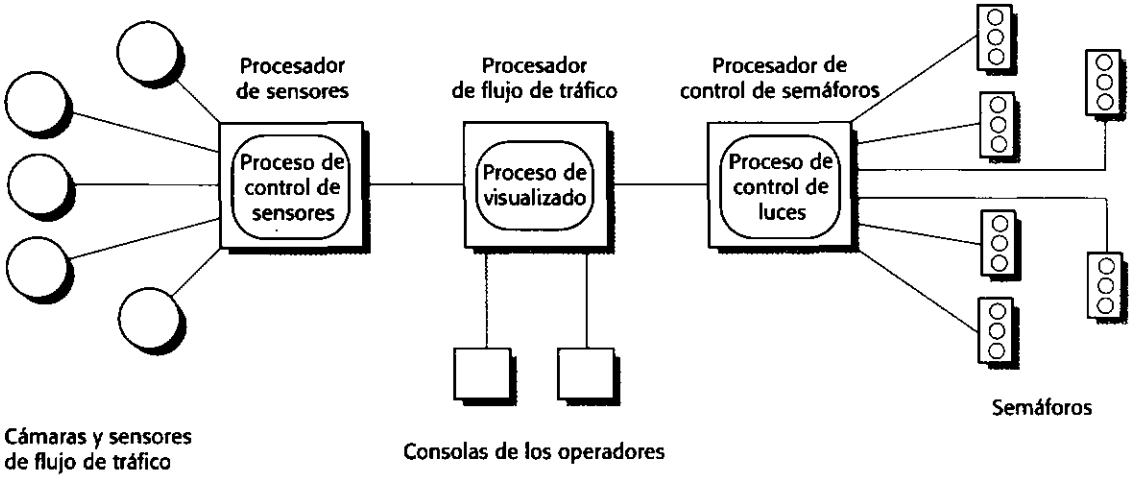


Figura 12.1 Un sistema multiprocesador de control de tráfico.

Los sistemas software compuestos de múltiples procesos no son necesariamente sistemas distribuidos. Si se dispone de más de un procesador, entonces se puede implementar la distribución, pero los diseñadores del sistema no siempre consideran forzosamente cuestiones de distribución durante el proceso de diseño. La aproximación de diseño para este tipo de sistemas es esencialmente la misma que para sistemas de tiempo real, tal y como se explica en el Capítulo 15.

12.2 Arquitecturas cliente-servidor

Ya hemos introducido el concepto de arquitecturas cliente-servidor en el Capítulo 11. En una arquitectura cliente-servidor, una aplicación se modela como un conjunto de servicios proporcionados por los servidores y un conjunto de clientes que usan estos servicios (Orfali y Harkey, 1998). Los clientes necesitan conocer qué servidores están disponibles, pero normalmente no conocen la existencia de otros clientes. Clientes y servidores son procesos diferentes, como se muestra en la Figura 12.2, que representa un modelo lógico de una arquitectura distribuida cliente-servidor.

Varios procesos servidores pueden ejecutarse sobre un único procesador servidor; por lo tanto, no hay necesariamente una correspondencia 1:1 entre procesos y procesadores en el sistema. La Figura 12.3 muestra la arquitectura física de un sistema con seis computadoras cliente y dos computadoras servidor. Éstos pueden ejecutar los procesos cliente y servidor mostrados en la Figura 12.2. Cuando hacemos referencia a *clientes* y *servidores*, nos referimos a los procesos lógicos en vez de a las computadoras físicas sobre las que se ejecutan.

El diseño de sistemas cliente-servidor debería reflejar la estructura lógica de la aplicación que se está desarrollando. Una forma de ver una aplicación se ilustra en la Figura 12.4, que muestra una aplicación estructurada en tres capas. La capa de presentación está relacionada con la presentación de la información al usuario y con toda la interacción con él. La capa de procesamiento de la aplicación está relacionada con la implementación de la lógica de la aplicación, y la capa de gestión de datos está relacionada con todas las operaciones sobre la base de datos. En los sistemas centralizados, estas capas no es necesario que estén claramente separadas. Sin embargo, cuando se está diseñando un sistema distribuido,

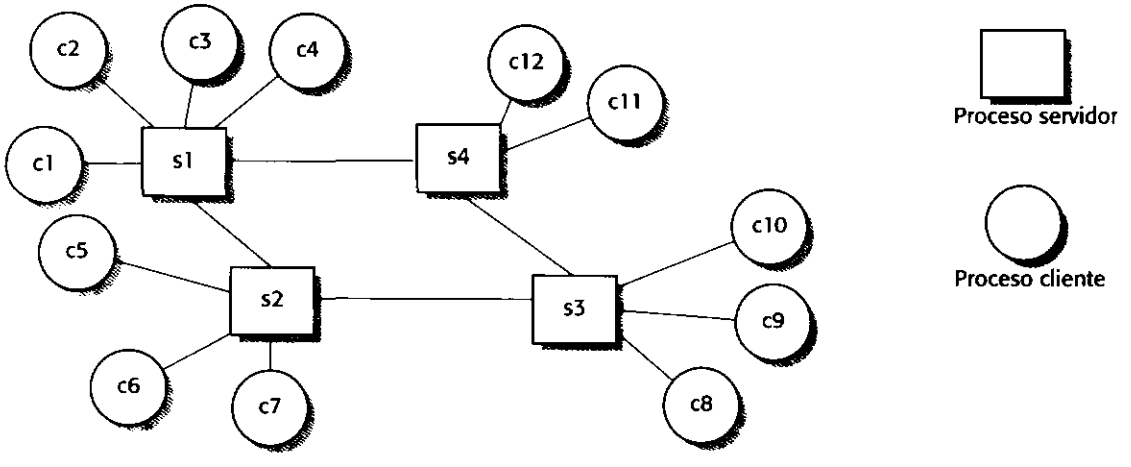


Figura 12.2 Un sistema cliente-servidor.

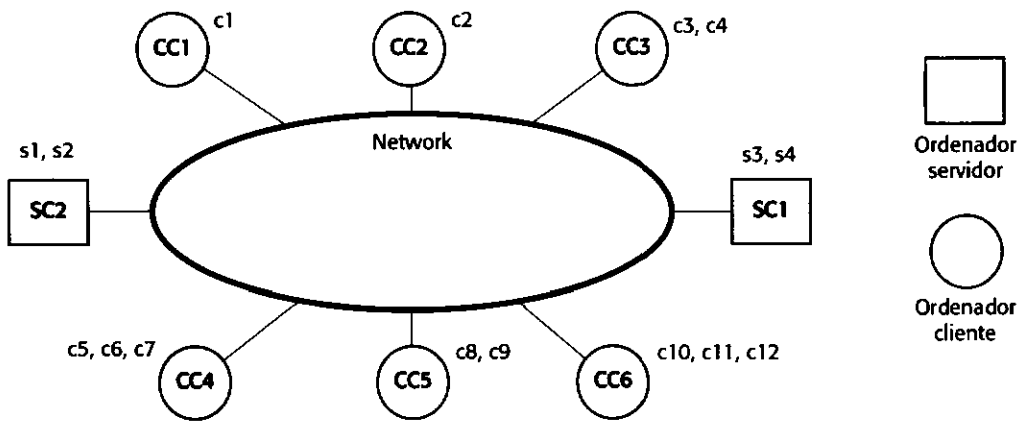


Figura 12.3 Computadoras en una red cliente-servidor.

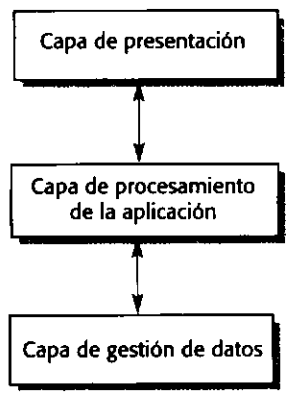


Figura 12.4 Capas de las aplicaciones.

debería hacerse una clara distinción entre ellas, de forma que sea posible distribuir cada capa sobre una computadora diferente.

La arquitectura cliente-servidor más simple se denomina arquitectura cliente-servidor de dos capas, en la que una aplicación se organiza como un servidor (o múltiples servidores idénticos) y un conjunto de clientes. Como se ilustra en la Figura 12.5, las arquitecturas cliente-servidor de dos capas pueden ser de dos tipos:

1. *Modelo de cliente ligero (thin-client).* En un modelo de cliente ligero, todo el procesamiento de las aplicaciones y la gestión de los datos se lleva a cabo en el servidor. El cliente simplemente es responsable de la capa de presentación del software.

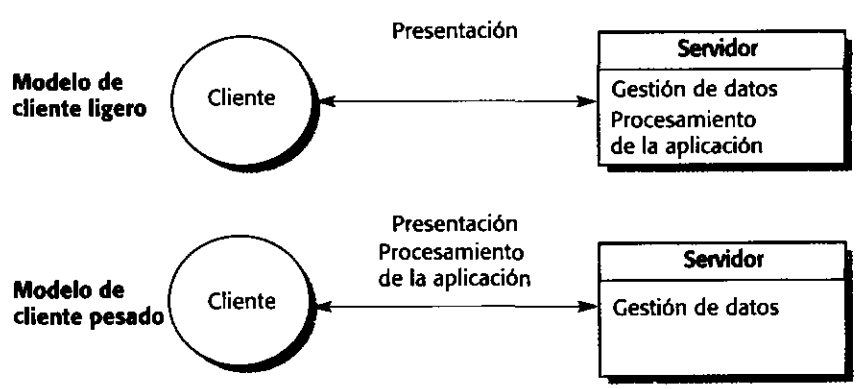


Figura 12.5 Clientes ligeros y ricos.

2. *Modelo de cliente rico (fat-client).* En este modelo, el servidor solamente es responsable de la gestión de los datos. El software del cliente implementa la lógica de la aplicación y las interacciones con el usuario del sistema.

Una arquitectura de dos capas con clientes ligeros es la aproximación más simple que se utiliza cuando los sistemas heredados centralizados, como se indica en el Capítulo 2, evolucionan a una arquitectura cliente-servidor. La interfaz de usuario para estos sistemas se migra a PCs, y la aplicación en sí misma actúa como un servidor y maneja todo el procesamiento de la aplicación y gestión de datos. Un modelo de cliente ligero también puede implementarse cuando los clientes son dispositivos de red sencillos en lugar de PCs o estaciones de trabajo. El dispositivo de red ejecuta un navegador de Internet y la interfaz de usuario es implementada a través de ese sistema.

Una gran desventaja del modelo de cliente ligero es que ubica una elevada carga de procesamiento tanto en el servidor como en la red. El servidor es responsable de todos los cálculos, y esto puede implicar la generación de un tráfico significativo en la red entre el cliente y el servidor. Los dispositivos de computación modernos disponen de una gran cantidad de potencia de procesamiento, la cual es bastante poco usada en la aproximación de cliente ligero.

El modelo de cliente rico hace uso de esta potencia de procesamiento disponible y distribuye tanto el procesamiento de la lógica de la aplicación como la presentación al cliente. El servidor es esencialmente un servidor de transacciones que gestiona todas las transacciones de la base de datos. Un ejemplo de este tipo de arquitectura es la de los sistemas bancarios ATM, en donde cada ATM es un cliente y el servidor es un *mainframe* que procesa la cuenta del cliente en la base de datos. El hardware de los cajeros automáticos realiza una gran cantidad de procesamiento relacionado con el cliente y asociado a la transacción.

Este sistema distribuido ATM se ilustra en la Figura 12.6. Observe que los ATMs no se conectan directamente con la base de datos de clientes sino con un monitor de teleproceso. Un monitor de teleproceso o gestor de transacciones es un sistema middleware que organiza las comunicaciones con los clientes remotos y serializa las transacciones de los clientes para su procesamiento en la base de datos. El uso de transacciones serializadas significa que el sistema puede recuperarse de los defectos sin corromper los datos del sistema.

Aunque el modelo de cliente rico distribuye el procesamiento de forma más efectiva que un modelo de cliente ligero, la gestión del sistema es más compleja. La funcionalidad de la aplicación se expande entre varias computadoras. Cuando la aplicación software tiene que ser modificada, esto implica la reinstalación en cada computadora cliente. Esto puede significar un coste importante si hay cientos de clientes en el sistema.

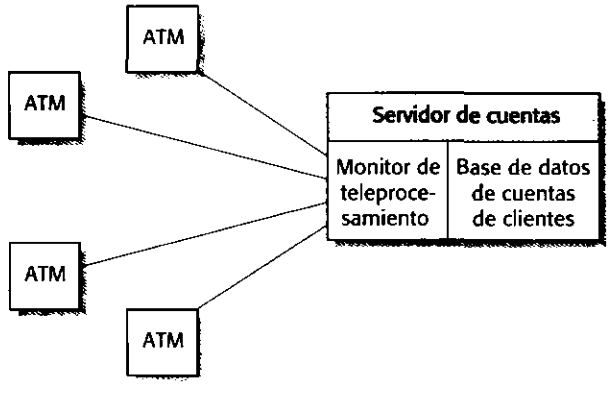


Figura 12.6
Un sistema ATM cliente-servidor.

La aparición del código móvil (como los applets de Java y los controles Active X), que pueden descargarse en un cliente desde un servidor, ha permitido el desarrollo de sistemas cliente-servidor que son algo intermedio entre los modelos de cliente ligero y rico. Algunas de las aplicaciones de procesamiento de software pueden descargarse en el cliente como código móvil, aligerando así la carga en el servidor. La interfaz de usuario se crea usando un navegador web que incluye utilidades de construcción de programas para ejecutar el código descargado.

El problema con una aproximación cliente-servidor de dos capas es que las tres capas lógicas —presentación, procesamiento de la aplicación y gestión de los datos— deben asociarse con dos computadoras el cliente y el servidor. Aquí puede haber problemas con la escalabilidad y rendimiento si se elige el modelo de cliente ligero, o problemas con la gestión del sistema si se usa el modelo de cliente rico. Para evitar estos problemas, una aproximación alternativa es usar una arquitectura *cliente-servidor de tres capas* (Figura 12.7). En esta arquitectura, la presentación, el procesamiento de la aplicación y la gestión de los datos son procesos lógicamente separados que se ejecutan sobre procesadores diferentes.

Un sistema bancario por Internet (Figura 12.8) es un ejemplo de una arquitectura cliente-servidor de tres capas. La base de datos de clientes del banco (usualmente ubicada sobre una computadora mainframe) proporciona servicios de gestión de datos; un servidor web proporciona los servicios de aplicación tales como facilidades para transferir efectivo, generar estados de cuenta, pagar facturas, y así sucesivamente; y la propia computadora del usuario con un navegador de Internet es el cliente. El sistema es escalable debido a que es relativamente fácil añadir nuevos servidores web a medida que el número de clientes crece.

El uso de una arquitectura de tres capas en este caso permite optimizar la transferencia de información entre el servidor web y el servidor de la base de datos. Las comunicaciones entre estos sistemas pueden usar protocolos de comunicación de bajo nivel muy rápidos. Para manejar la recuperación de información de la base de datos se utiliza un middleware eficiente que soporte consultas a la base de datos en SQL (Structured Query Language).

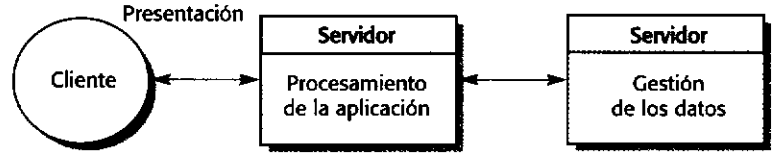


Figura 12.7
Una arquitectura cliente-servidor de tres capas.

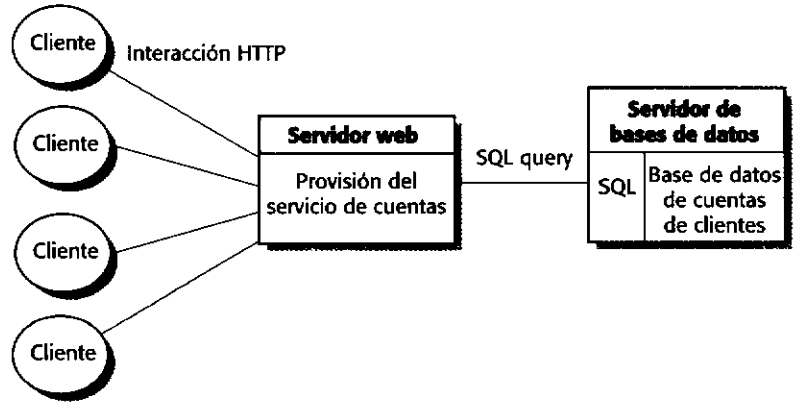


Figura 12.8
La arquitectura de distribución de un sistema bancario en Internet.

Arquitectura C/S de dos capas con clientes ligeros	Aplicaciones de sistemas heredados en donde no es práctico separar el procesamiento de la aplicación y la gestión de los datos. Aplicaciones que requieren cálculos intensivos tales como compiladores con poca o ninguna gestión de los datos. Aplicaciones que requieran manejar una gran cantidad de datos (navegar y consultar) con poco o ningún procesamiento de la aplicación.
Arquitectura C/S de dos capas con clientes ricos	Aplicaciones en donde el procesamiento de la aplicación se proporciona por software comercial (por ejemplo, Microsoft Excel) sobre el cliente. Aplicaciones que requieren un procesamiento de datos computacionalmente intensivo (por ejemplo, visualización de datos). Aplicaciones con una funcionalidad para el usuario final relativamente estable usada en un entorno de gestión del sistema bien establecido.
Arquitectura C/S de tres capas o multicapa	Aplicaciones a gran escala con cientos o miles de clientes. Aplicaciones en donde tanto los datos como la aplicación son volátiles. Aplicaciones en donde se integran datos de múltiples fuentes.

Figura 12.9 Uso de diferentes arquitecturas cliente-servidor.

En algunos casos resulta adecuado extender el modelo cliente-servidor de tres capas a una variante multicapa en la que se añaden al sistema servidores adicionales. Los sistemas multicapa pueden usarse cuando las aplicaciones necesitan acceder y usar datos de diferentes bases de datos. En este caso, un servidor de integración se ubica entre el servidor de aplicaciones y los servidores de la base de datos. El servidor de integración recoge los datos distribuidos y los presenta a la aplicación como si se obtuviesen desde una única base de datos.

Las arquitecturas cliente-servidor de tres capas y las variantes multicapa que distribuyen el procesamiento de la aplicación entre varios servidores son intrínsecamente más escalables que las arquitecturas de dos capas. El tráfico de la red se reduce al contrario que con las arquitecturas de dos capas de cliente ligero. El procesamiento de la aplicación es la parte más volátil del sistema, y puede ser fácilmente actualizada debido a que está localizada centralmente. El procesamiento, en algunos casos, puede distribuirse entre la lógica de la aplicación y los servidores de gestión de datos, en cuyo caso permite una respuesta más rápida a las peticiones de los clientes.

Los diseñadores de las arquitecturas cliente-servidor deben tener en cuenta una serie de factores cuando eligen la arquitectura más adecuada. En la Figura 12.9 se muestran las situaciones en las cuales las arquitecturas cliente-servidor analizadas aquí son probablemente las más adecuadas.

12.3 Arquitecturas de objetos distribuidos

En el modelo cliente-servidor de un sistema distribuido, los clientes y los servidores son diferentes. Los clientes reciben servicios de los servidores y no de otros clientes; los servidores pueden actuar como clientes recibiendo servicios de otros servidores, pero sin solicitar servicios de clientes; los clientes deben conocer los servicios que ofrece cada uno de los servidores y deben conocer cómo contactar con cada uno de estos servidores.

Este modelo funciona bien para muchos tipos de aplicaciones. Sin embargo, limita la flexibilidad de los diseñadores del sistema ya que ellos deben decidir dónde se proporciona cada

servicio. También deben planificar la escalabilidad y proporcionar algún medio para distribuir la carga sobre los servidores cuando más clientes se añadan al sistema.

Una aproximación más general al diseño de sistemas distribuidos es eliminar la distinción entre cliente y servidor y diseñar la arquitectura del sistema como una arquitectura de objetos distribuidos. En una arquitectura de objetos distribuidos (Figura 12.10), los componentes fundamentales del sistema son objetos que proporcionan una interfaz a un conjunto de servicios que ellos suministran. Otros objetos realizan llamadas a estos servicios sin hacer ninguna distinción lógica entre un cliente (el receptor de un servicio) y un servidor (el proveedor de un servicio).

Los objetos pueden distribuirse a través de varias computadoras en una red y comunicarse a través de middleware. A este middleware se lo denomina intermediario de peticiones de objetos. Su misión es proporcionar una interfaz transparente entre los objetos. Proporciona un conjunto de servicios que permiten la comunicación entre los objetos y que éstos sean añadidos y eliminados del sistema. En la Sección 12.3.1 se tratan los intermediarios de peticiones de objetos.

Las ventajas del modelo de objetos distribuido son las siguientes:

- Permite al diseñador del sistema retrasar decisiones sobre dónde y cómo deberían proporcionarse los servicios. Los objetos que proporcionan servicios pueden ejecutarse sobre cualquier nodo de la red. Por lo tanto, la distinción entre los modelos de cliente rico y ligero es irrelevante, ya que no hay necesidad de decidir con antelación dónde se sitúa la lógica de aplicación de los objetos.
- Es una arquitectura de sistema muy abierta que permite añadir nuevos recursos si es necesario. Como se indica en la siguiente sección, se han desarrollado e implementado estándares de comunicación de objetos que permiten escribir objetos en diferentes lenguajes de programación para comunicarse y proporcionar servicios entre ellos.
- El sistema es flexible y escalable. Se pueden crear diferentes instancias del sistema proporcionando los mismos servicios por objetos diferentes o por objetos reproducidos para hacer frente a las diferentes cargas del sistema. Pueden añadirse nuevos objetos a medida que la carga del sistema se incrementa sin afectar al resto de los objetos del sistema.
- Si es necesario, es posible reconfigurar el sistema de forma dinámica mediante la migración de objetos a través de la red. Esto puede ser importante donde haya fluctuación en los patrones de demanda de servicios. Un objeto que proporciona servicios puede mi-

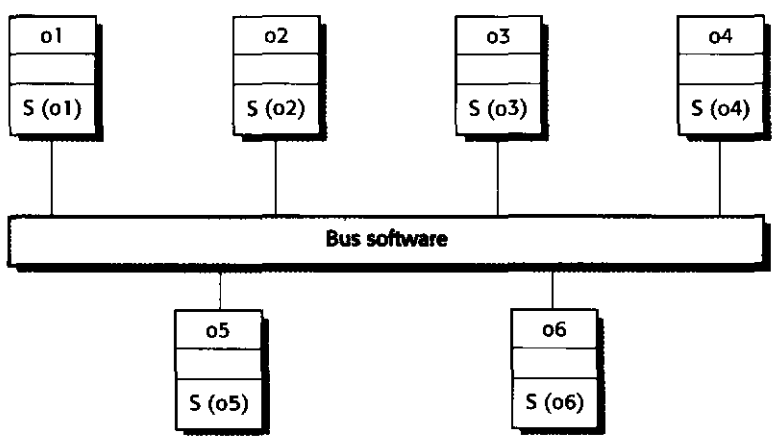


Figura 12.10 Arquitectura de objetos distribuidos.

grar al mismo procesador que los objetos que demandan los servicios, en lo que mejora el rendimiento del sistema.

Una arquitectura de objetos distribuidos puede ser usada como un modelo lógico que permita estructurar y organizar el sistema. En este caso se debe pensar en cómo proporcionar las funcionalidades de la aplicación únicamente en términos de servicios y combinaciones de servicios. A continuación se debe identificar cómo proporcionar estos servicios utilizando varios objetos distribuidos. En este nivel, los objetos que se diseñan son normalmente de grano grueso (denominados algunas veces *objetos de negocio*) que proporcionan servicios específicos del dominio. Por ejemplo, en una aplicación de venta al por menor, puede haber objetos de negocio relacionados con el control de existencias, comunicaciones con el cliente, pedidos de productos y otros. Por supuesto, este modelo lógico puede llevarse a cabo como un modelo de implementación.

De forma alternativa, se puede usar una aproximación de objetos distribuidos para implementar sistemas cliente-servidor. En este caso, el modelo lógico del sistema es un modelo cliente-servidor, pero tanto los clientes como los servidores se implementan como objetos distribuidos que se comunican a través de un bus software. Esto hace posible realizar cambios en el sistema de forma sencilla, por ejemplo, desde un sistema de dos capas a un sistema multicapa. En este caso, el servidor o el cliente puede no implementarse como un único objeto distribuido, sino que puede estar compuesto por objetos más pequeños que proporcionan servicios especializados.

Un ejemplo de un tipo de sistema en el que una arquitectura de objetos distribuidos podría ser adecuada es un sistema de minería de datos que busca relaciones entre los datos almacenados en varias bases de datos (Figura 12.11). Un ejemplo de aplicación de minería de datos podría ser un negocio de venta al por menor que tuviese, por ejemplo, tiendas de comestibles y tiendas de ferretería, y quisiera encontrar las relaciones entre compras de diversos tipos de comestibles y de ferretería. Por ejemplo, la gente que quiera comprar comida para bebé también puede comprar un tipo concreto de papel para las paredes. Con este conocimiento, la empresa podría entonces ofrecer ofertas específicas a los clientes de comida para bebé combinables con otras.

En este ejemplo, cada base de datos puede encapsularse como un objeto distribuido con una interfaz que proporciona acceso de sólo lectura a sus datos. Los objetos integradores se ocupan cada uno de ellos de tipos específicos de relaciones, y recogen información desde to-

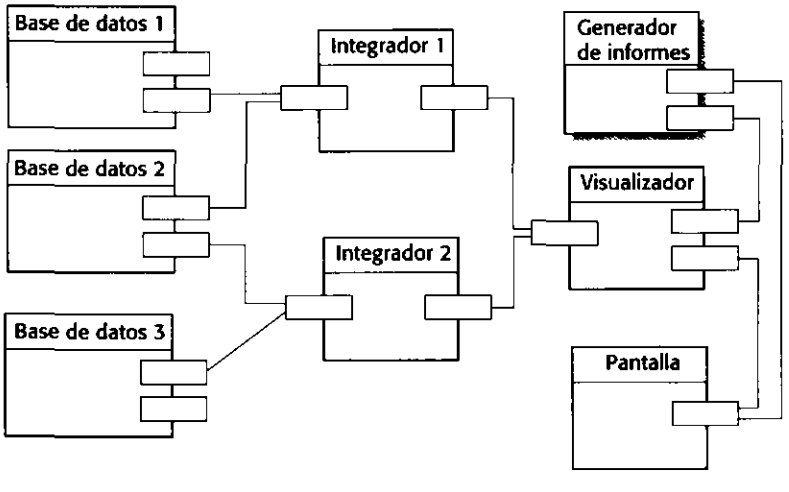


Figura 12.11 La arquitectura de distribución de un sistema de minería de datos.

das las bases de datos para intentar deducir dichas relaciones. Podría haber un objeto integrador que se ocupara de las variaciones de ventas de comestibles de temporada y otro que se ocupara de las relaciones entre los diferentes tipos de comestibles.

Los objetos visualizadores interaccionan con los objetos integradores para producir una visualización o un informe de las relaciones descubiertas. Debido a que se manejan grandes volúmenes de datos, los objetos visualizadores usarán normalmente representaciones gráficas de las relaciones que hayan descubierto. Se aborda la presentación de la información gráfica en el Capítulo 16.

Una arquitectura de objetos distribuidos es adecuada para este tipo de aplicación en lugar de una aplicación cliente-servidor por tres razones:

1. A diferencia de un sistema bancario ATM (por ejemplo), el modelo lógico del sistema no es el de provisión de servicios en el que se pueden distinguir servicios de gestión de datos.
2. Se pueden añadir bases de datos al sistema sin mayor problema. Cada base de datos es simplemente otro objeto distribuido. Los objetos de bases de datos pueden proporcionar una interfaz simplificada que controle el acceso a los datos. Las bases de datos a las que se puede acceder pueden residir en diferentes máquinas.
3. Permite explorar nuevos tipos de relaciones añadiendo nuevos objetos integradores. Las partes del negocio que estén interesadas en relaciones específicas pueden extender el sistema añadiendo objetos integradores que operen sobre sus computadoras sin requerir conocimiento de ningún otro integrador que se use en cualquier otra parte del sistema.

La principal desventaja de las arquitecturas de objetos distribuidos es que son mucho más complejas de diseñar que los sistemas cliente-servidor. Los sistemas cliente-servidor parecen ser la forma más natural de concebir a los sistemas. Éstos reflejan muchas transacciones humanas en las que la gente solicita y recibe servicios de otra gente especializada en proporcionar dichos servicios. Es más difícil pensar en una provisión de servicios generales, y todavía no tenemos una gran experiencia en el diseño y desarrollo de objetos de negocio de grano grueso.

12.3.1 CORBA

Tal y como se ha indicado en la sección previa, la implementación de una arquitectura de objetos distribuidos requiere un middleware (intermediarios de peticiones de objetos) para gestionar las comunicaciones entre los objetos distribuidos. En principio, los objetos en el sistema pueden implementarse utilizando diferentes lenguajes de programación, los objetos pueden ejecutarse sobre diferentes plataformas y sus nombres no necesitan ser conocidos por el resto de los objetos del sistema. Por lo tanto, el «pegamento» middleware tiene que realizar mucho trabajo para asegurar la transparencia en las comunicaciones de los objetos.

Se requiere middleware a dos niveles para soportar la computación de objetos distribuidos:

1. A nivel de comunicación lógica, el middleware proporciona funcionalidades que permiten a los objetos intercambiar datos y controlar la información sobre diferentes computadoras. Se han desarrollado estándares como CORBA y COM (Pritchard, 1999) para facilitar las comunicaciones lógicas de objetos sobre diferentes plataformas.
2. A nivel de componentes, el middleware proporciona una base para desarrollar componentes compatibles. Estándares tales como componentes CORBA, EJB o Active X

(Szyperski, 2002) proporcionan una base para la implementación de componentes con métodos estándar que pueden requerirse y usarse por otros componentes. Comentamos los estándares de componentes en el Capítulo 19.

En esta sección, se trata el middleware para comunicaciones lógicas de objetos y se explica cómo se soporta esto por los estándares CORBA. Estos estándares fueron definidos por el *Object Management Group* (OMG), que define los estándares para el desarrollo orientado a objetos. Los estándares OMG están disponibles, de forma gratuita, desde su sitio web.

La visión de OMG de una aplicación distribuida se muestra en la Figura 12.12, que se ha adaptado del diagrama de Siegel de la *Object Management Architectura* (Siegel, 1998). Ésta propone que una aplicación distribuida debería estar formada por varios componentes:

1. Objetos de aplicación diseñados e implementados para esta aplicación.
2. Objetos estándar definidos por el OMG para un dominio específico. Estos estándares para objetos del dominio incluyen áreas como financieras/aseguradoras, comercio electrónico y médicas.
3. Los servicios fundamentales CORBA, que proporcionan servicios de computación distribuida tales como gestión de seguridad y directorios.
4. Facilidades CORBA horizontales tales como facilidades de interfaz de usuario, facilidades para gestión del sistema, y otras. La denominación *facilidades horizontales* sugiere que estas facilidades son comunes a muchos dominios de aplicación y, por lo tanto, se usan en muchas aplicaciones diferentes.

Los estándares CORBA incluyen todos los aspectos de esta visión. Hay cuatro elementos principales para estos estándares:

1. Un modelo de objetos para objetos de aplicación en donde un objeto CORBA es una encapsulación de un estado con una interfaz con un lenguaje neutral y bien definido descrito en IDL (*Interface Definition Language*).
2. Un intermediario de peticiones de objetos (ORB) que gestiona peticiones para servicios de objetos. Este ORB localiza al objeto que proporciona el servicio, lo prepara para la petición, envía la petición de servicio y devuelve el resultado al solicitante del servicio.
3. Un conjunto de servicios de objetos que son servicios generales que probablemente serán requeridos por muchas aplicaciones distribuidas. Ejemplos de servicios son servicios de directorio, servicios de transacciones y servicios de persistencia.

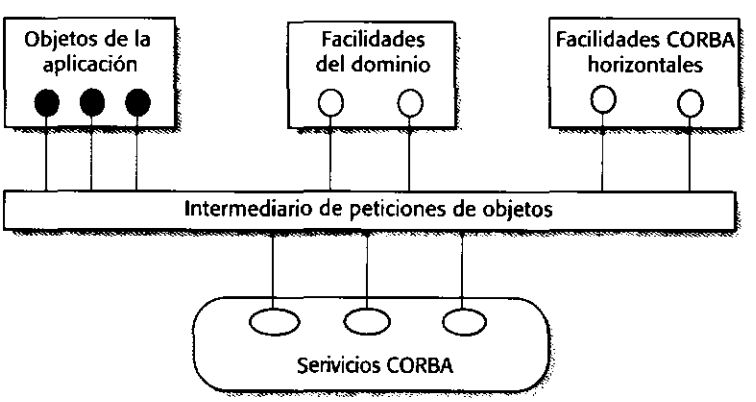


Figura 12.12 La estructura de una aplicación distribuida basada en CORBA.

4. Un conjunto de componentes comunes construidos sobre estos servicios básicos que pueden ser requeridos por las aplicaciones. Pueden ser componentes verticales específicos del dominio o componentes horizontales de propósito general usados por muchas aplicaciones.

El modelo de objetos CORBA considera que un objeto es una encapsulación de atributos y servicios, como ocurre con los objetos normales. Sin embargo, los objetos CORBA deben tener una definición de la interfaz separada que defina los atributos y operaciones públicas del objeto. Las interfaces de objetos CORBA se definen utilizando un lenguaje de definición de interfaces estándar independiente del lenguaje. Si un objeto desea usar los servicios proporcionados por otro objeto, entonces accede a dichos servicios a través de la interfaz IDL. Los objetos CORBA tienen un único identificador denominado referencia de objeto interoperable (IOR). Este IOR se usa cuando un objeto solicita los servicios de otro.

El intermediario de peticiones de objeto conoce a los objetos que están solicitando servicios y a sus interfaces. El ORB gestiona la comunicación entre los objetos. Los objetos que se comunican no necesitan conocer la localización de otros objetos ni necesitan conocer nada acerca de su implementación. Como la interfaz IDL aísla los objetos del ORB, es posible cambiar la implementación del objeto de una forma completamente transparente. La localización del objeto puede cambiar entre las invocaciones, lo cual es transparente para el resto de los objetos del sistema.

Por ejemplo, en la Figura 12.13, dos objetos o1 y o2 se comunican a través de un ORB. El objeto que hace la llamada (o1) tiene un *stub* IDL asociado que define la interfaz del objeto que proporciona el servicio solicitado. El implementador de o1 incluye la llamada a este stub en su implementación del objeto cuando se solicita un servicio. El IDL es un superconjunto de C++; por lo tanto, es muy fácil acceder a este stub si se está programando en C++, y es igualmente fácil hacerlo en C o Java. Las correspondencias entre otros lenguajes e IDL también han sido definidas para lenguajes como, por ejemplo, ADA y COBOL.

El objeto que proporciona el servicio tiene un *skeleton* IDL asociado que enlaza la interfaz con la implementación de los servicios. En términos simples, cuando un servicio se llama a través de la interfaz, el skeleton IDL traduce esta petición en una llamada al servicio al lenguaje de implementación que se haya utilizado. Cuando el método o procedimiento sea ejecutado, el skeleton IDL traduce los resultados a IDL para que puedan ser accedidos por el objeto que realizó la llamada. Cuando un objeto proporciona servicios a otros objetos y también usa los servicios proporcionados en alguna parte, necesita ambos, un skeleton y un stub IDL. Se requiere un stub IDL para cada objeto que sea usado.

Los intermediarios de peticiones de objetos no se implementan normalmente como procesos separados, pero son un conjunto de librerías que pueden enlazarse con las implementa-

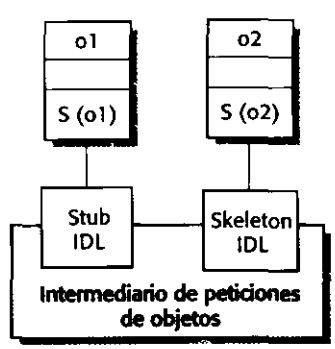


Figura 12.13 Comunicaciones de objetos a través de un ORB.

ciones de los objetos. Por lo tanto, en un sistema distribuido, cada computadora que está ejecutando objetos distribuidos tendrá su propio intermediario de peticiones de objetos. Éste manejará todas las invocaciones locales de objetos. Sin embargo, una petición de un servicio proporcionado por un objeto remoto requiere comunicaciones inter-ORB.

Esta situación se ilustra en la Figura 12.14. En este ejemplo, cuando el objeto o1 u o2 solicitan un servicio de o3 o de o4, los ORBs asociados deben comunicarse. Una implementación CORBA soporta comunicación ORB-a-ORB siempre que se proporcione a todos los ORBs el acceso a las definiciones de la interfaz IDL y además éstos implementen los estándares de OMG denominados Protocolo Genérico Inter-ORB (GIOP). Este protocolo define mensajes estándar que los ORBs pueden intercambiar para implementar llamadas a objetos remotos y transferir información. Cuando GIOP se combina con los protocolos de Internet TCP/IP, el GIOP permite a los ORBs comunicarse a través de Internet.

La iniciativa CORBA ha estado presente desde los años 80, y las primeras versiones de CORBA simplemente pretendían dar soporte a los objetos distribuidos. Sin embargo, a medida que los estándares han evolucionado, se han ido extendiendo cada vez más. Además de un mecanismo para comunicaciones de objetos distribuidos, los estándares CORBA definen actualmente algunos servicios estándar que pueden proporcionarse para soportar aplicaciones orientadas a objetos distribuidos.

Se puede pensar en los servicios CORBA como facilidades que probablemente sean requeridas por muchos sistemas distribuidos. Los estándares definen aproximadamente 15 servicios comunes. Algunos ejemplos de estos servicios genéricos son:

1. Servicios de nombres y de categorías (*trading*) que permiten a los objetos hacer referencia y encontrar a otros objetos de la red. El servicio de nombres es un servicio de directorios que permite a los objetos asignar nombres a otros objetos y encontrarlos. Esto es similar a las páginas blancas de un listín telefónico. Los servicios trading son como las páginas amarillas. Los objetos pueden encontrar a otros objetos que se hayan registrado con este servicio y acceder a la especificación de dichos objetos.
2. Servicios de notificación que permiten a los objetos notificar a otros objetos que ha ocurrido algún evento. Los objetos pueden registrar su interés en un evento específico del servicio y, cuando el evento ocurre, éste se les notifica automáticamente. Por ejemplo, supongamos que el sistema incluye una cola de impresión que encola documentos para ser impresos y a varios objetos impresora. La cola de impresión registra que está interesada en un evento de «final de impresión» de un objeto impresora. El servicio de notificación informa al objeto interesado cuando la impresión se

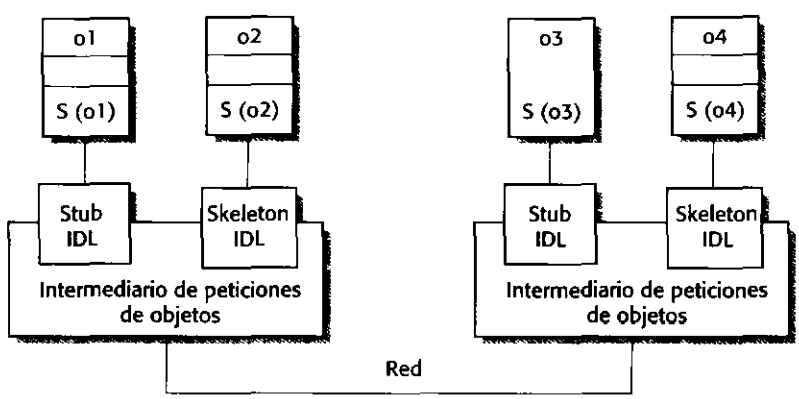


Figura 12.14 Comunicaciones inter-ORB.

completa. A continuación éste puede programar el siguiente documento sobre esa impresora.

3. Servicios de transacciones que soportan transacciones atómicas y operaciones *rollback* cuando ocurre un fallo. Las transacciones son una facilidad tolerante a defectos que soportan la recuperación de errores durante una operación de actualización. Si una operación de actualización de un objeto falla, entonces el estado del objeto puede volver a su estado anterior a la actualización (operación *rollback*).

Los estándares CORBA incluyen definiciones de interfaz para un amplio rango de componentes horizontales y verticales que pueden usarse por los constructores de aplicaciones distribuidas. Los componentes verticales son componentes específicos para un dominio de aplicación. Los componentes horizontales son componentes de propósito general como los componentes de interfaz de usuario. El desarrollo de especificaciones para componentes CORBA horizontales y verticales es una actividad a largo plazo, y las especificaciones disponibles actualmente son publicadas en el sitio web de OMG.

12.4 Computación distribuida interorganizacional

Por razones de seguridad e interoperabilidad, la computación distribuida ha sido principalmente implementada a nivel organizacional. Una organización tiene varios servidores y reparte su carga computacional entre ellos. Debido a que dichos servidores están todos dentro de la misma organización, pueden aplicarse estándares locales y procesos operacionales. Aunque, para los sistemas basados en web, las computadoras cliente están a menudo fuera de los límites de la organización, su funcionalidad está limitada a la ejecución de software de interfaz de usuario.

Sin embargo, actualmente están disponibles modelos más recientes de computación distribuida que permiten computación distribuida interorganizacional en lugar de intraorganizacional. Comentamos dos de estas aproximaciones en esta sección. La computación peer-to-peer se basa en cálculos que se llevan a cabo en nodos individuales de la red. Los sistemas orientados a servicios están relacionados con los servicios distribuidos en lugar de con objetos distribuidos, y también se relacionan con estándares basados en XML para intercambio de datos.

12.4.1 Arquitecturas peer-to-peer

Los sistemas peer-to-peer (p2p) son sistemas descentralizados en los que los cálculos pueden llevarse a cabo en cualquier nodo de la red y, al menos en principio, no se hacen distinciones entre clientes y servidores. En las aplicaciones peer-to-peer, el sistema en su totalidad se diseña para aprovechar la ventaja de la potencia computacional y disponibilidad de almacenamiento a través de una red de computadoras potencialmente enorme. Los estándares y protocolos que posibilitan las comunicaciones a través de los nodos están embebidos en la propia aplicación, y cada nodo debe ejecutar una copia de dicha aplicación.

En el momento de escribir este libro, las tecnologías peer-to-peer han sido mayormente usadas para sistemas personales (Oram, 2001). Por ejemplo, los sistemas de compartición de ficheros basados en los protocolos Gnutella y Kazaa se usan para compartir ficheros sobre PCs de usuario, y sistemas de mensajería instantánea tales como ICQ y Jabber proporcionan comunicaciones directas entre usuarios sin un servidor intermedio. SETI@home es un proyec-

to a largo plazo para procesar datos de radiotelescopios sobre PCs desde casa para buscar indicios de vida extraterrestre, y Freenet es una base de datos descentralizada que ha sido diseñada para hacer más fácil la publicación de información de forma anónima y hacer que sea más difícil para las autoridades suprimir esta información.

Sin embargo, hay indicios de que esta tecnología se está utilizando de forma creciente en empresas para que las redes de PCs soporten la potencia de dichas empresas (McDougall, 2000). Intel y Boeing han implementado sistemas p2p para aplicaciones que requieren computaciones intensivas. Para aplicaciones cooperativas que soportan trabajo distribuido, ésta parece ser la tecnología más efectiva.

Usted puede ver la arquitectura de las aplicaciones p2p desde dos puntos de vista. La arquitectura lógica de la red es la distribución de la arquitectura del sistema, mientras que la arquitectura de la aplicación es la organización genérica de los componentes en cada tipo de aplicación. Este capítulo se centra en los dos tipos principales de arquitecturas lógicas de red que se pueden usar: arquitecturas descentralizadas y arquitecturas semicentralizadas.

En principio, en los sistemas peer-to-peer cada nodo en la red podría conocer cualquier otro nodo, podría conectarse con él, y podría intercambiar datos. En la práctica, por supuesto, esto es imposible, ya que los nodos se organizan dentro de «localidades» con algunos nodos que actúan como puentes a otras localidades de nodos. La Figura 12.15 muestra esta arquitectura p2p descentralizada.

En una arquitectura descentralizada, los nodos en la red no son simplemente elementos funcionales, sino que también son interruptores de comunicaciones que pueden encaminar los datos y señales de control de un nodo a otro. Por ejemplo, supongamos que la Figura 12.15 representa un sistema de gestión de documentos descentralizado. El sistema es usado por un consorcio de investigadores para compartir documentos, y cada miembro del consorcio mantiene su propio almacén de documentos. Sin embargo, cuando un documento es recuperado, el nodo que recupera ese documento también lo hace disponible para los otros nodos. Cualquiera que necesite un documento genera un comando de búsqueda que es enviado a los nodos de esa «localidad». Estos nodos comprueban si tienen el documento y, si es así, lo devuelven al que ha hecho la petición. Si dichos nodos no tienen el documento, encaminan la búsqueda a otros nodos; cuando finalmente se encuentra el documento, el nodo puede encaminarlo de vuelta al nodo original que hizo la petición. Por lo tanto, si n1 emite una búsqueda de un documento que está almacenado en n10, esta búsqueda se encamina a través de los nodos n3, n6 y n9 hasta llegar a n10.

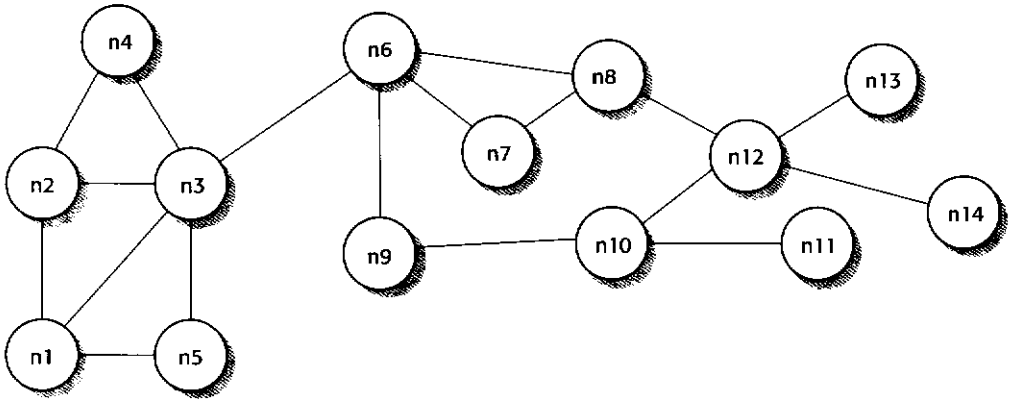


Figura 12.15 Arquitectura p2p descentralizada.

Esta arquitectura descentralizada tiene ventajas obvias en tanto que es altamente redundante, y por lo tanto es tolerante a defectos y tolerante a nodos desconectados de la red. Sin embargo, existen sobrecargas obvias en el sistema ya que la misma búsqueda puede ser procesada por muchos nodos diferentes y hay una sobrecarga significativa en comunicaciones entre iguales replicadas. Un modelo de arquitectura p2p alternativo que parte de una arquitectura p2p pura es una arquitectura semicentralizada en la que, dentro de la red, uno o más nodos actúan como servidores para facilitar las comunicaciones entre los nodos. La Figura 12.16 ilustra este modelo.

En una arquitectura semicentralizada, el papel del servidor es ayudar a establecer contacto entre iguales en la red o para coordinar los resultados de un cálculo. Por ejemplo, si la Figura 12.16 representa un sistema de mensajería instantánea, entonces los nodos de la red se comunican con el servidor (indicado por líneas discontinuas), para encontrar qué otros nodos están disponibles. Una vez que éstos son encontrados, se pueden establecer comunicaciones directas y la conexión con el servidor es innecesaria. Por lo tanto, los nodos n2, n3, n5 y n6 están en comunicación directa.

En un sistema computacional p2p en donde un cálculo que requiere un uso intensivo del procesador se distribuye a través de un gran número de nodos, es normal que se distingan algunos nodos cuyo papel es distribuir el trabajo a otros nodos y reunir y comprobar los resultados del cálculo.

Si bien hay sobrecargas obvias en los sistemas peer-to-peer, éstos son una aproximación mucho más eficiente para la computación interorganizacional que la aproximación basada en servicios que se comentan en la siguiente sección. Todavía hay problemas en el uso de las aproximaciones p2p para computación interorganizacional, ya que cuestiones tales como protección y autenticidad siguen todavía sin resolverse. Esto significa que los sistemas p2p son más probables de usar en sistemas de información no críticos o bien cuando ya existen relaciones de trabajo entre las organizaciones.

12.4.2 Arquitectura de sistemas orientados a servicios

El desarrollo de la WWW trajo consigo que las computadoras cliente tuviesen acceso a los servidores remotos situados fuera de sus propias organizaciones. Si estas organizaciones convertían su información a formato HTML, entonces ésta podía ser accedida por estas computadoras. Sin embargo, el acceso se realizaba solamente a través de un navegador web, y el acceso directo a los almacenes de información por otros programas no era práctico. Esto implicaba que las conexiones oportunistas entre servidores en donde, por ejemplo, un programa solicitaba información a varios catálogos, no eran posibles.

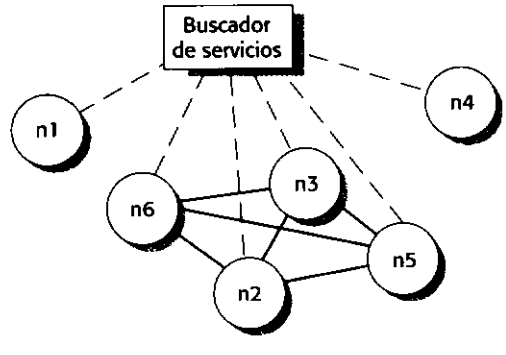


Figura 12.16 Una arquitectura p2p semicentralizada.

Para solucionar este problema, se propuso la noción de un servicio web. Mediante el uso de un servicio web, las organizaciones que quieren hacer accesible la información a otros programas, pueden hacerlo definiendo y publicando una interfaz de servicio web. Esta interfaz define los datos disponibles y cómo se puede acceder a ellos. De forma más general, un servicio web es una representación estándar para cualquier recurso computacional o de información que pueda ser usado por otros programas. Por lo tanto, se podría definir un servicio de recaudación de impuestos en el que los usuarios podrían rellenar sus formularios de impuestos y éstos ser automáticamente comprobados y enviados a las autoridades de hacienda.

Un servicio web es una instancia de una noción más general de un servicio, la cual se define en (Lovelock *et al.*, 1996) como:

> *un acto o realización ofertada por una de las partes a otra. Si bien el proceso puede estar asociado a un producto físico, la realización es esencialmente intangible, y no se convierte normalmente en propietaria de cualquiera de los factores de la producción.*

La esencia de un servicio, por lo tanto, es que la provisión de servicio es independiente de la aplicación que usa el servicio (Turner *et al.*, 2003). Los proveedores de servicios pueden desarrollar servicios especializados y ofertarlos a un cierto número de usuarios de servicios desde diferentes organizaciones. Las aplicaciones pueden construirse enlazando los servicios desde varios proveedores utilizando bien un lenguaje de programación estándar o bien un lenguaje de instrumentación de servicios tal como BPEL4WS.

Existen varios modelos de servicios, desde el modelo JINI (Kumaran, 2001) pasando por los servicios web (Stal, 2002) y servicios de rejilla (Foster *et al.*, 2002). Conceptualmente, todas ellos operan de acuerdo con el modelo mostrado en la Figura 12.17, la cual es una generalización del modelo conceptual de servicios web descrito por Kreger (Kreger, 2001). Un proveedor de servicio oferta un servicio definiendo su interfaz y definiendo la funcionalidad del servicio. Un solicitante del servicio enlaza este servicio en su aplicación. Esto significa que la aplicación del solicitante incluye código para llamar al servicio y procesa el resultado de la llamada al servicio. Para asegurar que el servicio puede ser accedido por usuarios externos a dicho servicio, el proveedor de servicios registra una entrada en el servicio de registro que incluye información sobre el servicio y lo que hace.

Las diferencias entre este modelo de servicios y la aproximación de objetos distribuidos para arquitecturas de sistemas distribuidos son las siguientes:

- Los servicios pueden ofertarse por cualquier proveedor de servicio dentro o fuera de una organización. Suponiendo que éstos cumplen ciertos estándares (analizados más adelante), las organizaciones pueden crear aplicaciones integrando servicios desde varios proveedores. Por ejemplo, un compañía de fabricación puede enlazar directamente a los servicios proporcionados por sus proveedores.

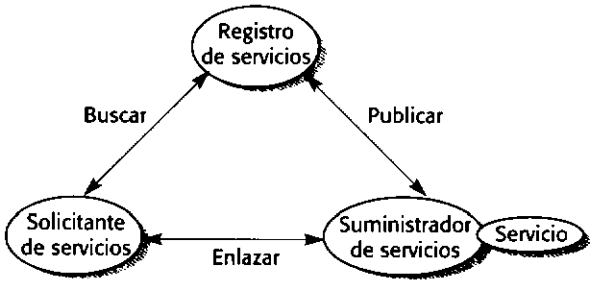


Figura 12.17 Arquitectura conceptual de un sistema orientado a servicios.

- El proveedor de servicios hace pública la información sobre el servicio para que cualquier usuario autorizado pueda usarlo. El proveedor de servicios y el usuario de los servicios no necesitan negociar sobre lo que el servicio hace antes de ser incorporado en un programa de aplicación.
- Las aplicaciones pueden retrasar el enlace de los servicios hasta que éstas sean desplegadas o hasta que estén en ejecución. Por lo tanto, una aplicación que usa un servicio de precios de productos (por ejemplo) podría cambiar dinámicamente los proveedores de los servicios mientras el sistema se está ejecutando.
- Es posible la construcción oportunista de nuevos servicios. Un proveedor de servicios puede reconocer nuevos servicios que se crean enlazando servicios existentes de formas innovadoras.
- Los usuarios de los servicios pueden pagar por los servicios con arreglo a su uso en vez de a su provisión. Por lo tanto, en lugar de comprar un componente de precio elevado que se usa muy raramente, el desarrollador de la aplicación puede usar un servicio externo por el que pagará solamente cuando sea requerido.
- Las aplicaciones pueden hacerse más pequeñas (lo cual es importante si tienen que estar embebidas en otros dispositivos) debido a que pueden implementar el manejo de excepciones como servicios externos.
- Las aplicaciones pueden ser reactivas y adaptar su operación de acuerdo con su entorno enlazando con diferentes servicios a medida que cambia éste.

En el momento de escribir este libro, estas ventajas han suscitado un gran interés en los servicios web como base para la construcción de aplicaciones distribuidas débilmente acopladas. Sin embargo, la experiencia práctica con las arquitecturas orientadas a servicios todavía es limitada, por lo que aún no conocemos las implicaciones prácticas de esta aproximación.

La reutilización del software ha sido un tema de investigación durante muchos años; todavía, tal y como se indica en los Capítulos 18 y 19, quedan pendientes muchas dificultades prácticas en la reutilización del software. Uno de los principales problemas ha sido que los estándares para componentes reutilizables han sido desarrollados hace relativamente poco tiempo, y varios de estos estándares están en uso. Sin embargo, la iniciativa de los servicios web ha estado guiada por estándares desde su nacimiento, y los estándares que incluyen muchos aspectos de los servicios web están desarrollándose. Los tres estándares fundamentales que permiten la comunicación entre servicios web son:

1. *SOAP* (*Simple Object Access Protocol*). Este protocolo define una organización para intercambio de datos estructurados entre servicios web.
2. *WSDL* (*Web Services Description Language*). Este protocolo define cómo pueden representarse las interfaces de servicios web.
3. *UDDI* (*Universal Description, Discovery and Integration*). Éste es un estándar de búsqueda que define cómo puede organizarse la información de descripción de servicios, usada por los solicitantes de los servicios para encontrar servicios.

Todos estos estándares se basan en XML, un lenguaje legible por los humanos y las máquinas (Skonnard y Gudgin, 2002). Sin embargo, no es necesario conocer detalles de estos estándares para comprender el concepto de servicios web.

Las arquitecturas de aplicaciones de servicios web son arquitecturas débilmente acopladas en donde el enlace a los servicios puede cambiar durante la ejecución. Algunos sistemas se construirán solamente usando servicios web y otros mezclarán servicios web desarrollados localmente. Para ilustrar cómo pueden organizarse las aplicaciones, considere la siguiente situación:

Un sistema de información en un vehículo proporciona dispositivos con información sobre el tiempo, condiciones del tráfico de carretera, información local y otras. Éste se enlaza con la radio del vehículo para que la información sea proporcionada como una señal sobre un canal de radio específico. El vehículo es equipado con un receptor de GPS para descubrir su posición y, basándose en esa posición, el sistema accede a un cierto número de servicios de información. La información puede proporcionarse en el lenguaje especificado del dispositivo.

La Figura 12.18 ilustra una posible organización para el sistema anterior. El software del vehículo incluye cinco módulos. Éstos gestionan las comunicaciones con el dispositivo, con un receptor GPS que informa de la posición del vehículo y con la radio del vehículo. Los módulos **Transmisor** y **Receptor** gestionan todas las comunicaciones con los servicios externos.

El vehículo se comunica con un servicio de información móvil proporcionado externamente que añade información desde otros servicios que proporcionan información sobre el tiempo, tráfico y facilidades locales. Diferentes proveedores en distintos lugares proporcionan este servicio, y el sistema software del vehículo usa un servicio de búsqueda para localizar el servicio de información adecuado y enlazar con él. El servicio de búsqueda también es usado por el servicio de información móvil para enlazar con los servicios adecuados de tiempo, tráfico y facilidades. Los servicios intercambian mensajes SOAP que incluyen la información de la posición GPS usada, para seleccionar la información adecuada. La información añadida se devuelve al vehículo a través de un servicio que traduce el lenguaje de información al lenguaje del dispositivo.

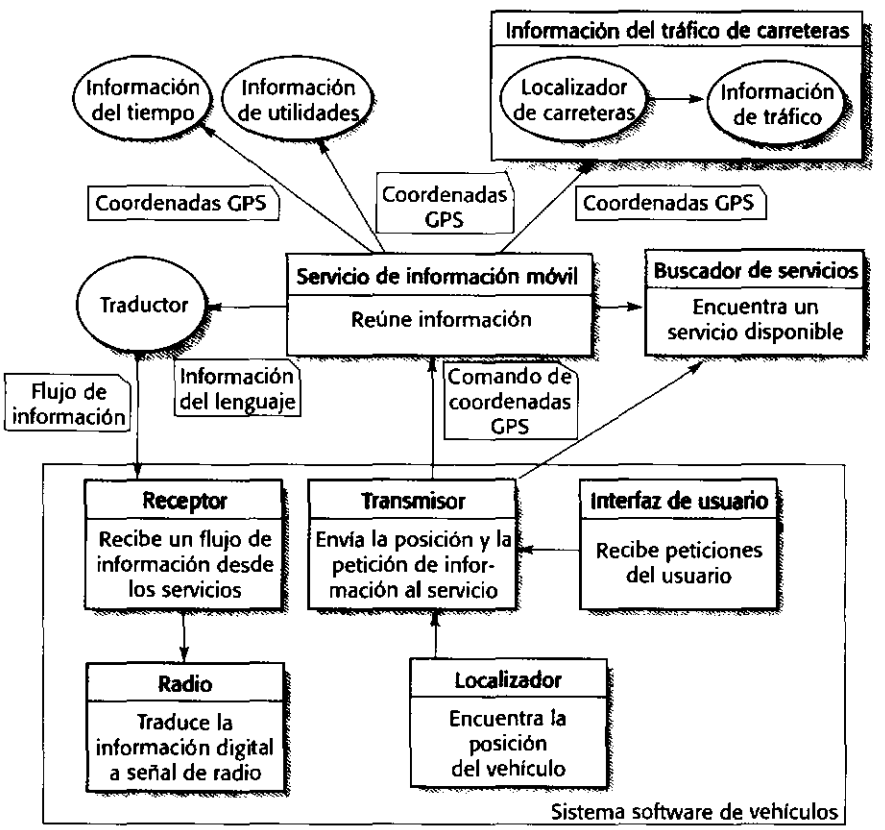


Figura 12.18 Un sistema de información de vehículo basado en servicios.

Este ejemplo ilustra una de las ventajas clave de la aproximación orientada a servicios. No es necesario decidir cuándo se programa o despliega el sistema, qué suministrador de servicio debería usarse y a qué servicios específicos se debería acceder. A medida que el vehículo se mueve, el software del vehículo usa el servicio de búsqueda para encontrar el servicio de información más adecuado y enlazar con él. Debido al uso de un servicio de traducción, la información puede traspasar los límites locales y, por lo tanto, hacer que la información local esté disponible para gente que no hable el lenguaje local.

Esta visión de computación orientada a servicios todavía no es realizable con los actuales servicios web, en los que el enlazado de servicios a las aplicaciones todavía es bastante estático. Sin embargo, antes de que aparezca una nueva edición de este libro, es probable que haya más enlaces dinámicos y arquitecturas de aplicaciones. No hay duda de que el modelo orientado a servicios representa una forma nueva muy importante de implementar los sistemas de computación distribuidos interorganizacionales.

PUNTOS CLAVE

- **Los sistemas distribuidos permiten compartir recursos, son abiertos, concurrentes, escalables, tolerantes a defectos y transparentes.**
- **Los sistemas cliente-servidor son sistemas distribuidos en los que el sistema se modela como un conjunto de servicios proporcionados por servidores a procesos cliente.**
- **En un sistema cliente-servidor, la interfaz de usuario siempre se ejecuta en el cliente, y la gestión de datos siempre es proporcionada por un servidor compartido. La funcionalidad de la aplicación puede ser implementada en la computadora cliente o en el servidor.**
- **En una arquitectura de objetos distribuidos, no hay distinción entre clientes y servidores. Los objetos proporcionan servicios generales que pueden ser llamados por otros objetos. Esta aproximación puede ser usada para implementar sistemas cliente-servidor.**
- **Los sistemas de objetos distribuidos requieren un software intermediario (middleware) para gestionar las comunicaciones de objetos y para permitir que los objetos se puedan añadir y eliminar del sistema.**
- **Los estándares CORBA son un conjunto de estándares para middleware que soportan arquitecturas de objetos distribuidos. Incluyen definiciones de modelos de objetos, definiciones de un intermediario de peticiones de objetos y definiciones de servicios comunes. Están disponibles varias implementaciones de los estándares CORBA.**
- **Las arquitecturas peer-to-peer son arquitecturas descentralizadas en las que no se distinguen clientes y servidores. Los cálculos pueden distribuirse sobre muchos sistemas en organizaciones diferentes.**
- **Los sistemas orientados a servicios se crean enlazando servicios software proporcionados por varios suministradores de servicios. Un aspecto importante de las arquitecturas orientadas a servicios es que los componentes arquitectónicos pueden retrasarse hasta que el sistema es desplegado o está en ejecución.**

LECTURAS ADICIONALES

«Turning software into a service». Un buen artículo que analiza los principios de la computación orientada a servicios. A diferencia de muchos artículos sobre este tema, no encubre estos principios con un análisis de los estándares relacionados. [M. Turner *et al.*, *IEEE Computer*, 36 (10), octubre 2003.]

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Aunque este libro no trata en profundidad las arquitecturas p2p, es una excelente introducción a la computación p2p y analiza la organización y aproximación usadas en varios sistemas p2p. [A. Oram (ed.), 2001, O'Reilly and Associates, Inc.]

Distributed Systems: Concepts and Design, 3rd ed. Un libro de texto fácil de entender que discute todos los aspectos del diseño e implementación de sistemas distribuidos. Los dos primeros capítulos son particularmente relevantes para el capítulo aquí expuesto. (G. Couloris *et al.*, 2001, Addison-Wesley.)

«Middleware: A model for distributed systems services». Éste es un excelente artículo que resume el papel del middleware en los sistemas distribuidos y examina los distintos servicios middleware que se pueden proporcionar. [P. A. Bernstein, *Comm. ACM*, 39 (2), febrero 1996).

EJERCICIOS

12.1 Explique por qué los sistemas distribuidos son intrínsecamente más escalables que los sistemas centralizados. ¿Cuáles son los límites más probables de la escalabilidad del sistema?

12.2 ¿Cuál es la diferencia fundamental entre una aproximación de cliente rico y una de cliente ligero para el desarrollo de sistemas cliente-servidor? Explique por qué el uso de Java como lenguaje de implementación atenúa la diferencia entre estas aproximaciones.

12.3 Su cliente quiere desarrollar un sistema para almacenar información en donde los distribuidores pueden acceder a la información de las compañías, y pueden evaluar varios escenarios de inversión usando un sistema de simulación. Cada distribuidor usa esta simulación de forma diferente, de acuerdo con su experiencia y el tipo de almacenes con los que trabaja. Sugiera una arquitectura cliente-servidor para este sistema que muestre dónde se localiza la funcionalidad. Justifique el modelo cliente-servidor que usted ha elegido.

12.4 Haciendo referencia al modelo de aplicación mostrado en la Figura 12.4, comente los problemas que podrían tener lugar al convertir un sistema heredado de la década de los 80 que utiliza un mainframe para procesamiento de pólizas de seguros a una arquitectura cliente-servidor.

12.5 ¿Cuáles son las facilidades básicas que puede proporcionar un intermediario de peticiones de objetos?

12.6 Explique por qué el uso de objetos distribuidos con un intermediario de peticiones de objetos simplifica la implementación de sistemas cliente-servidor escalables. Ilustre su respuesta con un ejemplo.

12.7 ¿Cómo se usa el IDL de CORBA para soportar comunicaciones entre objetos que han sido implementados en diferentes lenguajes de programación? Explique por qué esta aproximación puede ocasionar problemas de rendimiento si hay diferencias sustanciales entre los lenguajes usados para la implementación de los objetos.

12.8 Usando una aproximación de objetos distribuidos, proponga una arquitectura para un sistema nacional de venta de entradas para teatros en donde los usuarios puedan comprobar la disponibilidad de las localidades y reservar éstas en un grupo de teatros. El sistema debería soportar devoluciones de entradas para que la gente pueda devolver sus entradas y que éstas se puedan vender a otros clientes en el último momento.

12.9 Indique dos ventajas y dos desventajas de las arquitecturas peer-to-peer descentralizadas y semicentralizadas.

12.10 ¿Cuáles son las ventajas del enlace dinámico en un sistema orientado a servicios?

12.11 Para el sistema de información del vehículo, explique por qué es mejor que el software del vehículo se comunique con un servicio global en lugar de hacerlo directamente con servicios de información individuales.

12.12 El desarrollo de la computación orientada a servicios se ha basado en la especificación y adopción temprana de estándares. Coemente el papel general de la estandarización para soportar y restringir la competición e innovación en el mercado del software.

13 Arquitecturas de aplicaciones

Objetivos

El objetivo de este capítulo es introducir varios modelos arquitectónicos para clases específicas de sistemas de aplicaciones software. Cuando haya leído este capítulo:

- conocerá dos organizaciones arquitectónicas fundamentales de los sistemas de negocio, denominados procesamiento por lotes y procesamiento transaccional;
- comprenderá la arquitectura abstracta de los sistemas de información y de gestión de recursos;
- comprenderá cómo los sistemas dirigidos por comandos, tales como los editores, pueden estructurarse como sistemas de procesamiento de eventos;
- conocerá la estructura y organización de los sistemas de procesamiento de lenguajes.

Contenidos

Tal y como se ha explicado en el Capítulo 11, se pueden ver las arquitecturas de los sistemas desde varias perspectivas. Hasta ahora, los análisis de las arquitecturas de sistemas en los Capítulos 11 y 12 se han centrado en perspectivas arquitectónicas y cuestiones tales como control, distribución y estructuración del sistema. En este capítulo, sin embargo, se utiliza una aproximación alternativa y se contemplan las arquitecturas desde la perspectiva de una aplicación.

Los sistemas de aplicaciones intentan adecuarse a necesidades organizacionales o de negocio. Todos los negocios tienen mucho en común —necesitan contratar a personas, emitir facturas, mantener la contabilidad y así sucesivamente— y esto es especialmente cierto para negocios que operan en el mismo sector. Por lo tanto, además de las funciones de negocio generales, todas las compañías telefónicas necesitan sistemas para conectar las llamadas, gestionar su red, emitir facturas a los clientes, etcétera. Como consecuencia, las aplicaciones de los sistemas que utilizan estos negocios tienen también mucho en común.

Normalmente, los sistemas del mismo tipo tienen arquitecturas similares, y la diferencia entre estos sistemas está en la funcionalidad detallada que proporcionan. Esto puede ilustrarse por el crecimiento de los sistemas de Planificación de Recursos de Empresas (ERP) tales como el sistema SAP/R3 (Appelrath y Ritter, 2000) y paquetes verticales de software para aplicaciones particulares. En estos sistemas, que se tratan brevemente en el Capítulo 18, un sistema genérico se configura y adapta para crear una aplicación específica de negocio. Por ejemplo, un sistema para suministrar una gestión de una cadena puede adaptarse para diferentes tipos de suministradores, artículos y acuerdos contractuales.

En el análisis que se hace aquí de las arquitecturas de aplicaciones, se presentan modelos estructurales genéricos de varios tipos de aplicaciones. Se estudia la organización básica de estos tipos de aplicaciones y, donde resulta apropiado, se descompone la arquitectura de alto nivel para mostrar los subsistemas que se incluyen normalmente en las aplicaciones.

Como diseñador de software, usted puede usar arquitecturas de aplicaciones genéricas de varias formas:

1. *Como un punto de partida para el proceso de diseño arquitectónico.* Si no está familiarizado con este tipo de aplicación, puede basar sus diseños iniciales sobre arquitecturas genéricas. Por supuesto, éstas tendrán que especializarse para sistemas concretos, pero son un buen punto de partida para su diseño.
2. *Como una lista de comprobación de un diseño.* Si ha desarrollado un diseño arquitectónico de un sistema, puede comprobarlo contrastándolo con la arquitectura de aplicación genérica para ver si ha omitido algún componente de diseño importante.
3. *Como una forma de organizar el trabajo del grupo de desarrollo.* La arquitectura de la aplicación identifica características estructurales estables de las arquitecturas de los sistemas y, en muchos casos, es posible desarrollarlas en paralelo. Usted puede asignar trabajo a miembros del grupo para implementar diferentes subsistemas dentro de la arquitectura.
4. *Como una forma de evaluar los componentes para su reutilización.* Si usted tiene componentes, podría ser capaz de reutilizarlos; puede comparar éstos con las estructuras genéricas para ver si es probable la reutilización en la aplicación que está desarrollando.
5. *Como un vocabulario para hablar sobre tipos de aplicaciones.* Si está tratando una aplicación específica o intentando comparar aplicaciones del mismo tipo, entonces puede usar los conceptos identificados en la arquitectura genérica para hablar de las aplicaciones.

Hay muchos tipos de sistemas de aplicaciones y, aparentemente, pueden parecer muy distintos. Sin embargo, cuando se examina la organización arquitectónica de las aplicaciones, muchas de estas aplicaciones aparentemente distintas tienen mucho en común. Ilustramos esto describiendo las arquitecturas de cuatro grandes tipos de aplicaciones:

1. *Aplicaciones de procesamiento de datos.* Las aplicaciones de procesamiento de datos son aplicaciones conducidas por los datos. Procesan datos por lotes sin intervenciones explícitas del usuario durante el procesamiento. Las acciones específicas tomadas por la aplicación dependen de los datos que se están procesando. Los sistemas de procesamiento por lotes se usan normalmente en aplicaciones de negocio en donde se realizan operaciones similares sobre grandes cantidades de datos. Dichas aplicaciones manejan un amplio rango de funciones administrativas tales como nóminas, facturación, contabilidad y publicidad.
2. *Aplicaciones de procesamiento de transacciones.* Las aplicaciones de procesamiento de transacciones son aplicaciones centradas en bases de datos que procesan peticiones del usuario para obtener información y para actualizar la información en una base de datos. Éstos son los tipos más comunes de sistemas de negocio interactivos. Se organizan de forma que las acciones del usuario no pueden interferir entre sí y se mantenga la integridad de la base de datos. Esta clase de sistema incluye sistemas bancarios interactivos, sistemas de comercio electrónico, sistemas de información y sistemas de reservas.
3. *Sistemas de procesamiento de eventos.* Ésta es una clase muy amplia de aplicaciones en las que las acciones del sistema dependen de la interpretación de eventos en el entorno del sistema. Estos eventos podrían ser la entrada de una orden por un usuario del sistema o un cambio en las variables que son monitorizadas por el sistema. Muchas aplicaciones basadas en PCs, entre las que se incluyen juegos, sistemas de edición tales como procesadores de texto, hojas de cálculo, editores de imágenes y sistemas de presentación son sistemas de procesamiento de eventos. Los sistemas de tiempo real, estudiados en el Capítulo 15, también se encuentran dentro de esta categoría.
4. *Sistemas de procesamiento de lenguajes.* Los sistemas de procesamiento de lenguajes son sistemas en los que las intenciones del usuario se expresan en un lenguaje formal (como, por ejemplo, Java). Los sistemas de procesamiento de lenguajes procesan este lenguaje en algún formato interno y entonces interpretan su representación interna. Los sistemas de procesamiento de lenguajes más conocidos son los compiladores, que traducen programas de lenguajes de alto nivel a código máquina. Sin embargo, los sistemas de procesamiento de lenguajes se utilizan también para interpretar lenguajes de comandos para bases de datos, sistemas de información y lenguajes de marcado tales como XML (Harold y Means, 2002), que se usa de forma amplia para describir elementos de datos estructurados.

Se han elegido estos tipos particulares de sistemas debido a que representan la mayoría de sistemas que se usan en la actualidad. Los sistemas de negocio son generalmente sistemas de procesamiento de transacciones o de datos, y la mayoría del software de computadoras personales se construye sobre una arquitectura de procesamiento de eventos. Los sistemas de tiempo real son también sistemas de procesamiento de eventos; estas arquitecturas se tratan en el Capítulo 15. Todo el desarrollo del software se centra en los sistemas de procesamiento de lenguajes tales como compiladores.

Los sistemas de procesamiento por lotes y procesamiento de transacciones se centran en bases de datos. Debido a la importancia fundamental de los datos, estos sistemas son comu-

nes para aplicaciones de diferentes tipos que comparten la misma base de datos. Por ejemplo, un sistema de procesamiento de datos de negocio que imprime situaciones bancarias usan la misma base de datos de clientes al igual que un sistema de procesamiento de transacciones que proporciona acceso basado en web para información de cuentas.

Por supuesto, tal y como se vio en el Capítulo 11, las aplicaciones complejas raramente siguen un único modelo arquitectónico. En su lugar, su arquitectura es la mayoría de las veces un híbrido, con diferentes partes de la aplicación estructuradas de forma diferente. Por lo tanto, al diseñar estos sistemas, hay que considerar las arquitecturas de subsistemas individuales y también cómo éstas tienen que integrarse con la arquitectura del sistema en su totalidad.

13.1 Sistemas de procesamiento de datos

Los negocios necesitan relacionarse con sistemas de procesamiento de datos para soportar muchos aspectos de sus negocios tales como el pago de salarios, el cálculo y la impresión de facturas, el mantenimiento de cuentas y la emisión de renovación para pólizas de seguros. Como su nombre indica, estos sistemas se centran en datos, y las bases de datos con las que se relacionan son normalmente varios órdenes de magnitud más grandes que los propios sistemas. Los sistemas de procesamiento de datos son sistemas de procesamiento por lotes en los que los datos son introducidos y extraídos por lotes a partir de un fichero o base de datos en lugar de ser introducidos y extraídos por un terminal de usuario. Estos sistemas seleccionan datos para el registro de entradas y, dependiendo del valor de los campos en los registros, realizan algunas acciones especificadas en el programa. Pueden volver a escribir el resultado del cálculo en la base de datos y formatear la entrada y la salida calculada para su impresión.

La arquitectura de los sistemas de procesamiento por lotes tiene tres componentes principales, tal y como se ilustra en la Figura 13.1. Un componente de entrada reúne entradas desde una o más fuentes. Un componente de procesamiento realiza cálculos utilizando estas entradas; y un componente de salida genera salidas para ser escritas en la base de datos e impresas. Por ejemplo, un sistema de facturas telefónicas toma los datos registrados de un cliente y las lecturas realizadas del teléfono (entradas) de una centralita telefónica, calcula los costes para cada cliente (proceso) y entonces imprime facturas (salidas) para cada cliente.

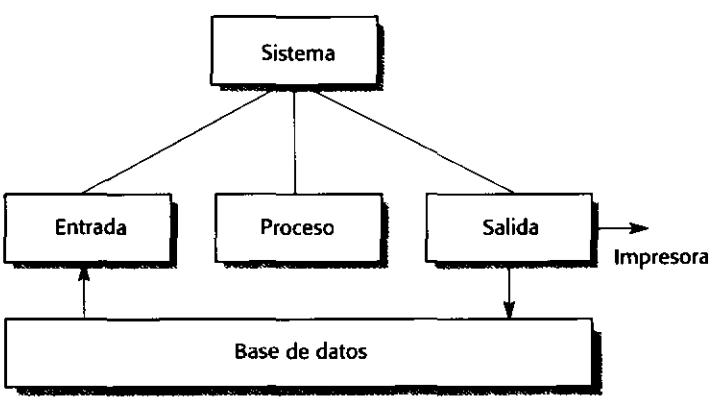


Figura 13.1
Un modelo de entrada-proceso-salida de un sistema de procesamiento de datos.

Los componentes de entrada, procesamiento y salida pueden descomponerse en una estructura de entrada-proceso-salida. Por ejemplo:

1. Un componente de entrada puede leer algún dato (entrada) desde un fichero o base de datos, comprobar la validez de los datos y corregir algunos errores (proceso), y a continuación encolar los datos válidos para su procesamiento (salida).
2. Un componente de procesamiento puede escoger una transacción de una cola (entrada), realizar algunos cálculos sobre los datos y crear un nuevo registro de datos almacenando los resultados de los cálculos (proceso), y a continuación encolar este nuevo registro para su impresión (salida). Algunas veces el procesamiento se realiza dentro de la base de datos del sistema y otras veces como un programa separado.
3. Un componente de salida puede leer registros de una cola (entrada), formatear éstos de acuerdo con el formato de salida (proceso), y a continuación enviarlos a una impresora o volver a escribir nuevos registros en la base de datos (salida).

La naturaleza de los sistemas de procesamiento de datos, en donde los registros o transacciones se procesan en serie sin necesidad de mantener el estado entre las transacciones, implica que estos sistemas sean naturalmente orientados a funciones en vez de orientados a objetos. Las funciones son componentes que no mantienen información del estado interno de una invocación a otra. Los diagramas de flujo de datos, introducidos en el Capítulo 8, son una buena manera de describir la arquitectura de los sistemas de procesamiento de datos de negocio.

Los diagramas de flujos de datos constituyen una forma de representar sistemas orientados a funciones en donde cada rectángulo con bordes redondeados en el flujo de datos representa una función que implementa algunas transformaciones de datos, y cada flecha representa un elemento de datos procesado por la función. Los ficheros o almacenes de datos se representan como rectángulos. La ventaja de los diagramas de flujo de datos es que muestran el procesamiento de los datos desde su entrada hasta su salida. Es decir, se pueden ver todas las funciones que actúan sobre los datos a medida que se mueven a través de las etapas del sistema. La estructura fundamental de un flujo de datos consiste en una función de entrada que pasa los datos a una función de procesamiento y a continuación a una función de salida.

La Figura 13.2 ilustra cómo pueden utilizarse los diagramas de flujo de datos para mostrar una visión más detallada de la arquitectura de un sistema de procesamiento de datos. Esta figura muestra el diseño de un sistema de pago de salarios. En este sistema, la información sobre los empleados en la organización es leída por el sistema, se calcula el salario mensual y las deducciones, y se realizan los pagos. Puede verse cómo este sistema sigue la estructura básica de entrada-proceso-salida:

1. Las funciones en la parte izquierda del diagrama **Lectura de registro de empleado**, **Lectura de datos a pagar mensualmente** y **Validación de datos de empleado** introducen los datos para cada empleado y comprueban dichos datos.
2. La función **Calcular salario** calcula el salario bruto para cada empleado y las deducciones a realizar sobre dicho salario. A continuación se calcula el salario neto mensual.
3. Las funciones de salida escriben una serie de ficheros que contienen detalles de las deducciones realizadas y el salario a pagar. Estos ficheros son procesados por otros programas una vez que se han calculado los detalles para todos los empleados. Finalmente el sistema imprime una nómina para el empleado que contiene el salario neto y las deducciones realizadas.

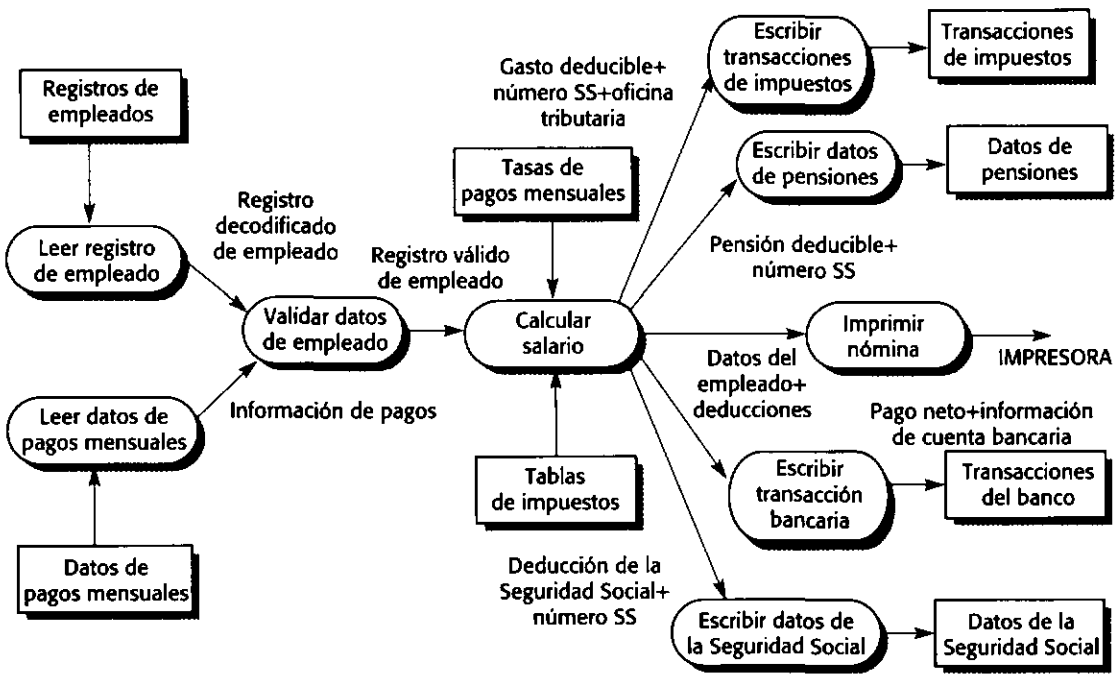


Figura 13.2 Diagrama de flujo de datos de un sistema de nóminas.

El modelo arquitectónico de los programas de procesamiento de datos es relativamente simple. Sin embargo, en estos sistemas la complejidad de la aplicación se refleja a menudo en los datos que se están procesando. Por lo tanto, el diseño de la arquitectura del sistema implica pensar o reflexionar sobre la arquitectura de los datos (Bracket, 1994) así como en la arquitectura del programa. El diseño de la arquitectura de los datos está fuera del ámbito de este libro.

13.2 Sistemas de procesamiento de transacciones

Los sistemas de procesamiento de transacciones se diseñan para procesar peticiones de usuario a fin de obtener información de una base de datos o peticiones para actualizar la base de datos (Lewis *et al.*, 2003). Técnicamente, una transacción de una base de datos es una secuencia de operaciones tratada como una única unidad (una unidad atómica). Todas las operaciones de una transacción tienen que ser completadas antes de que los cambios en la base de datos sean permanentes. Esto significa que los fallos de operaciones dentro de la transacción no conducen a inconsistencias en la base de datos.

Un ejemplo de una transacción es una petición de un cliente para efectuar un reintegro de una cuenta bancaria utilizando un ATM. Esto implica obtener detalles de la cuenta del cliente, comprobar el saldo, modificar el saldo por la cantidad reintegrada y enviar comandos al ATM para proporcionar el dinero en efectivo. Hasta que todos estos pasos no hayan sido completados, la transacción está incompleta y no se modifica la base de datos de cuentas de clientes.

Desde la perspectiva de un usuario, una transacción es cualquier secuencia coherente de operaciones que satisface un objetivo, tal como «encuentra los horarios de vuelos desde Londres a París». Si la transacción del usuario no requiere que la base de datos sea modificada,

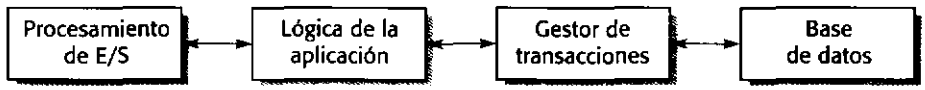


Figura 13.3
La estructura de las aplicaciones de procesamiento de transacciones.

entonces puede no ser necesario empaquetar estas operaciones como una transacción técnica de base de datos.

Los sistemas de procesamiento de transacciones son normalmente sistemas interactivos en donde los usuarios realizan peticiones de servicios de forma asíncrona. La Figura 13.3 ilustra la estructura arquitectónica de alto nivel de estas aplicaciones. Primero un usuario realiza una petición al sistema a través de un componente de procesamiento de E/S. La petición se procesa por alguna lógica específica de la aplicación. Una transacción se crea y se envía a un gestor de transacciones, el cual normalmente está embebido en el sistema de gestión de base de datos. Después de que el gestor de transacciones haya asegurado que la transacción se ha realizado correctamente, envía una señal a la aplicación indicando que el procesamiento ha finalizado.

La estructura entrada-proceso-salida que podemos observar en las aplicaciones de procesamiento de datos también se aplican a muchos sistemas de procesamiento de transacciones. Algunos de estos sistemas son versiones interactivas de los sistemas de procesamiento por lotes. Por ejemplo, por un lado los bancos dan entrada fuera de línea a todas las transacciones de clientes, y después ejecutan por la noche estas transacciones en un lote utilizando la base de datos de cuentas. Esta aproximación se ha reemplazado en la mayoría de los casos por sistemas interactivos basados en transacciones que actualizan las cuentas en tiempo real.

Un ejemplo de un sistema de procesamiento de transacciones es un sistema bancario que permite a los clientes consultar sus cuentas y extraer dinero de un ATM. El sistema está compuesto por dos subsistemas software que cooperan: el software del ATM y el software de procesamiento de cuentas en el servidor de la base de datos del banco. La entrada y salida de los subsistemas se implementa como software en el ATM, mientras que el subsistema de procesamiento está en el servidor de la base de datos del banco. La Figura 13.4 muestra la arquitectura de este sistema. Se ha añadido algún detalle al diagrama básico de entrada-proceso-salida para mostrar los componentes que pueden estar implicados en las actividades de entrada, procesamiento y salida. Deliberadamente, no se ha sugerido cómo interactúan estos componentes internos, ya que la secuencia de operaciones puede diferir de una máquina a otra.

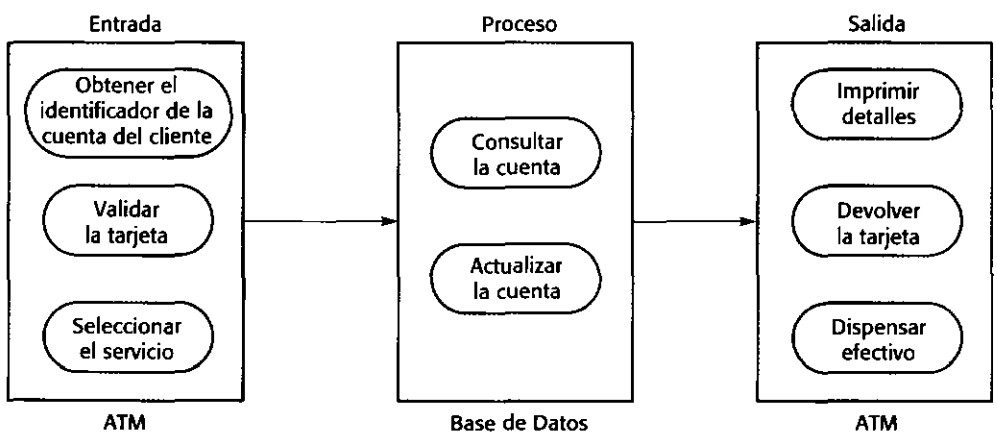


Figura 13.4
La arquitectura software de un ATM.

En sistemas tales como sistemas de cuentas de clientes bancarios, puede haber diferentes formas de interactuar con dicho sistema. Muchos clientes interactuarán a través de ATMs, pero el personal del banco utilizará terminales para acceder al sistema. Pueden utilizarse varios tipos de ATMs y terminales, y algunos clientes y personal pueden acceder a las cuentas a través de navegadores web.

Para simplificar la gestión de los diferentes protocolos de comunicación entre terminales, los sistemas de procesamiento de transacciones a gran escala pueden incluir middleware que comunica todos los tipos de terminales, organiza y serializa los datos desde los terminales, y envía los datos para su procesamiento. Este middleware, que analiza brevemente en el Capítulo 12, puede ser llamado por un monitor de teleprocesamiento o un sistema de gestión de transacciones. El sistema CICS de IBM (Horswill y Miller, 2000) es un ejemplo de dicho sistema muy extensamente usado.

La Figura 13.5 muestra otra visión de la arquitectura de un sistema de cuentas bancarias que maneja transacciones de cuentas personales desde los ATMs y terminales en un banco. El monitor de teleprocesamiento maneja la entrada y serializa las transacciones, que convierte en consultas a la base de datos. El procesamiento de las consultas tiene lugar en el sistema de gestión de base de datos. Los resultados se envían de vuelta al monitor de teleprocesamiento, el cual registra los terminales que han realizado la petición. A continuación este sistema organiza los datos en un formulario que puede ser manejado por el software del terminal y le devuelve los resultados de la transacción.

13.2.1 Sistemas de información y de gestión de recursos

Todos los sistemas que implican una interacción con una base de datos compartida pueden considerarse como sistemas de información basados en transacciones. Un sistema de información permite el acceso controlado a una gran base de información, como un catálogo de biblioteca, un horario de vuelos o los registros de pacientes en un hospital. El desarrollo de la WWW significa que un enorme número de sistemas de información pasaron de ser sistemas organizacionales especializados a ser sistemas de propósito general universalmente accesibles.

La Figura 13.6 es un modelo muy general de un sistema de información. El sistema se modela utilizando una aproximación de máquina abstracta o por capas (vista en la Sección 11.2.3), en donde la capa superior soporta la interfaz de usuario y la capa inferior la base de datos del sistema. La capa de comunicaciones con el usuario maneja todas las entradas y salidas de la interfaz de usuario, y la capa de recuperación de información incluye lógica específica de la aplicación para acceder y actualizar la base de datos. Tal y como veremos más

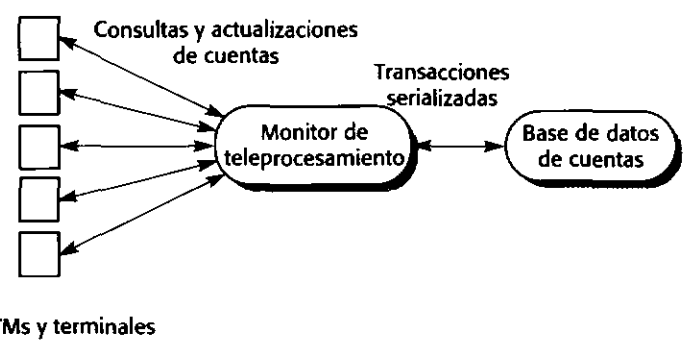


Figura 13.5 Middleware para gestión de transacciones.

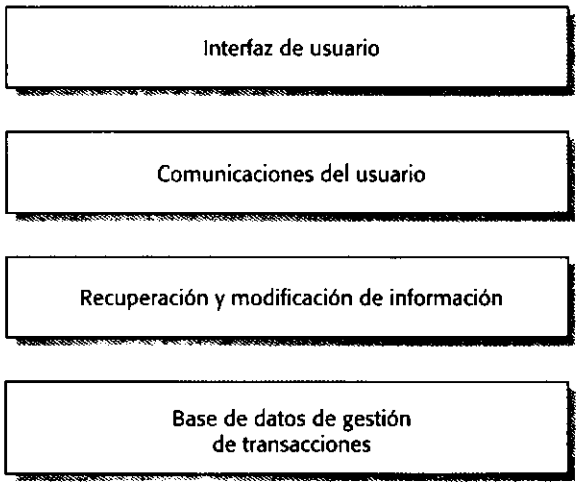


Figura 13.6
Un modelo por capas de un sistema de información.

adelante, las capas en este modelo pueden hacerse corresponder de forma directa con servidores en un sistema basado en Internet.

Como ejemplo de una instanciación de este modelo por capas, la Figura 13.7 presenta la arquitectura del sistema LIBSYS. Recuerde que este sistema permite a los usuarios acceder a documentos en bibliotecas remotas y descargarlos para su impresión. Se ha añadido detalle a cada capa en el modelo identificando los componentes que soportan comunicaciones con el usuario y el acceso y recuperación de información. Debería notarse también que la base de datos es una base de datos distribuida. Los usuarios realmente se conectan, a través del sistema, con las bases de datos de las bibliotecas que proporcionan los documentos.

La capa de comunicación con el usuario en la Figura 13.7 incluye tres componentes principales:

1. *El componente de identificación de usuario (login) LIBSYS* identifica y autentica a los usuarios. Todos los sistemas de información que restringen el acceso a un conjunto conocido de usuarios necesitan tener autenticación de usuario como una parte fundamental de sus sistemas de comunicación con el usuario. La autenticación de usuario

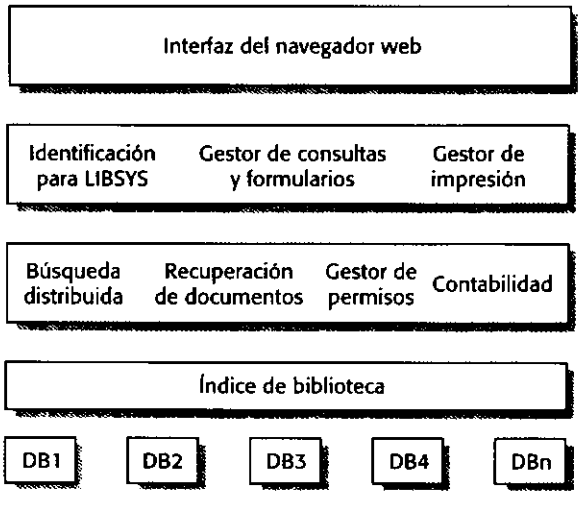


Figura 13.7
La arquitectura del sistema LIBSYS.

puede ser personal, pero en sistemas de comercio electrónico, puede también requerir que se proporcionen detalles sobre la tarjeta de crédito.

2. *El componente de gestión de consultas y formularios* gestiona los formularios que pueden presentarse al usuario y proporciona facilidades de consulta permitiendo al usuario solicitar información del sistema. De nuevo, los sistemas de información deben incluir un componente que proporcione estas facilidades.
3. *El componente de gestión de impresión* es específico de LIBSYS. Controla la impresión de documentos que, por razones de derechos de autor, puede estar restringida. Por ejemplo, algunos documentos solamente pueden imprimirse una vez en impresoras de la biblioteca registrada.

La capa de modificación y recuperación de información en el sistema LIBSYS incluye componentes específicos de la aplicación que implementan la funcionalidad del sistema. Estos componentes son:

1. *Búsqueda distribuida*. Este componente busca documentos como respuesta a consultas del usuario a través de todas las bibliotecas que están registradas en el sistema. La lista de bibliotecas conocidas se mantiene en el índice de bibliotecas.
2. *Recuperación de documentos*. Este componente recupera el documento o documentos que son solicitados por el usuario al servidor en el que se está ejecutando el sistema LIBSYS.
3. *Gestor de derechos*. Este componente maneja todos los aspectos de la gestión de derechos digitales y derechos de autor. Almacena un seguimiento de quién ha solicitado los documentos y, por ejemplo, asegura que no puedan realizarse múltiples peticiones para el mismo documento por la misma persona.
4. *Registro de cuentas*. Este componente registra todas las solicitudes y, si es necesario, maneja cualquier cargo que sea realizado por las bibliotecas en el sistema. También produce informes de gestión sobre el uso del sistema.

Nosotros podemos observar la misma estructura genérica de cuatro capas en otro tipo de sistema de información, como por ejemplo los sistemas diseñados para soportar la asignación de recursos. Los sistemas de asignación de recursos gestionan una cantidad fija de algún recurso determinado, como entradas para un concierto o un partido de fútbol. Éstos son asignados a los usuarios que solicitan dicho recurso al suministrador. Los sistemas de venta de entradas son un ejemplo obvio de un sistema de asignación de recursos, pero un gran número de programas aparentemente distintos son también realmente sistemas de asignación de recursos. Algunos ejemplos de este tipo de sistemas son:

1. *Sistemas de horarios* que asignan aulas a franjas horarias. El recurso a asignar aquí es un periodo de tiempo, y normalmente hay un gran número de restricciones con cada demanda del recurso.
2. *Sistemas de biblioteca* que gestionan el préstamo y retirada de libros u otros elementos. En este caso los recursos que se están asignando son los elementos que pueden ser prestados. En este tipo de sistemas, los recursos no son simplemente asignados, sino que algunas veces deben eliminarse las asignaciones hechas al usuario del recurso.
3. *Sistemas de gestión de tráfico aéreo* en donde el recurso que se está asignando es un sector del espacio aéreo para que se mantenga la separación entre los aviones que están siendo gestionados por el sistema. De nuevo, éste implica una asignación dinámica y reasignación del recurso, ya que el recurso es virtual en vez de un recurso físico.

Los sistemas de asignación de recursos son una clase de aplicaciones ampliamente utilizada. Si echamos un vistazo a su arquitectura con detalle, podemos ver cómo ésta se asemeja al modelo de sistema de información de la Figura 13.6. Los componentes de un sistema de asignación de recursos (mostrado en la Figura 13.8) comprenden:

1. Una *base de datos de recursos* que almacena detalles de los recursos que se están asignando. Los recursos pueden ser añadidos o eliminados de esta base de datos. Por ejemplo, en un sistema de biblioteca, la base de datos de recursos incluye detalles de todos los elementos que pueden ser prestados a los usuarios de la biblioteca. Normalmente esto se implementa utilizando un sistema de gestión de base de datos que incluye un sistema de procesamiento de transacciones. El sistema de gestión de base de datos también incluye facilidades para el bloqueo de recursos de forma que el mismo recurso no pueda ser asignado a usuarios que realizan solicitudes simultáneamente.
2. Un *conjunto de reglas* que describe las reglas para la asignación de recursos. Por ejemplo, un sistema de biblioteca normalmente limita a quién puede asignarse el recurso (usuarios registrados por la biblioteca), el periodo de tiempo que un libro u otro artículo puede ser prestado, el máximo número de libros que pueden prestarse, y así sucesivamente. Todo esto se encapsula en el componente de control de permisos de recursos.
3. Un *componente de gestión de recursos* que permite al suministrador de los recursos añadir, editar o borrar recursos del sistema.
4. Un *componente de asignación de recursos* que actualiza la base de datos de recursos cuando los recursos son asignados y asocia estos recursos con detalles del solicitante del recurso.
5. Un *módulo de autenticación de usuarios* que permite al sistema comprobar qué recursos están siendo asignados a un usuario acreditado. En un sistema de biblioteca, éste podría ser una tarjeta de biblioteca legible por una máquina; en un sistema de asignación de entradas, podría ser una tarjeta de crédito que verifica que el usuario es capaz de pagar el recurso.
6. Un *módulo de gestión de consultas* que permite a los usuarios consultar qué recursos están disponibles. En un sistema de biblioteca, esto podría basarse normalmente en consultas de elementos concretos; en un sistema de venta de entradas, podría implicar una visualización gráfica mostrando qué entradas están disponibles para fechas concretas.

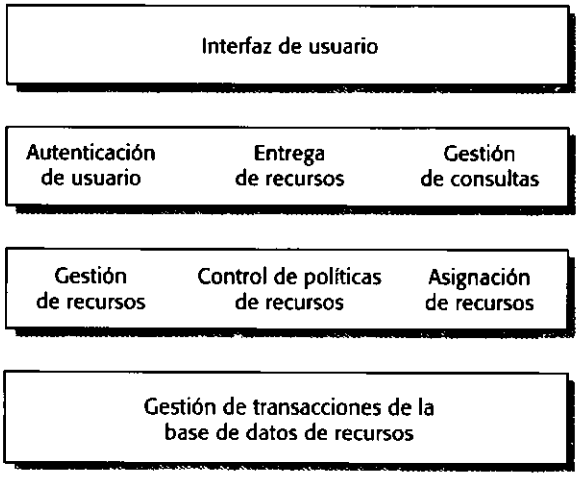


Figura 13.8
Un modelo por capas de un sistema de asignación de recursos.

7. Un *componente de entrega de recursos* que prepara los recursos para su entrega al solicitante. En un sistema de venta de entradas, esto podría implicar preparar un correo electrónico de confirmación y enviar una petición a una impresora de billetes para imprimir los billetes y los detalles de dónde deberían ser enviados.
8. Un *componente de interfaz de usuario* (a menudo un navegador web) que está fuera del sistema y permite al solicitante del recurso realizar consultas y peticiones para el recurso que se va a asignar.

Esta arquitectura por capas puede llevarse a cabo de varias formas. El software de sistemas de información puede organizarse para que cada capa sea un componente a gran escala que se ejecuta sobre un servidor distinto. Cada capa define sus interfaces externas y toda la comunicación tiene lugar a través de estas interfaces. De forma alternativa, si el sistema de información completo se ejecuta sobre una única computadora, entonces las capas intermedias se implementan normalmente como un único programa que se comunica con la base de datos a través de sus APIs. Una tercera alternativa consiste en implementar componentes de grano fino como distintos servicios web (discutidos en el Capítulo 12) y componer éstos de forma dinámica de acuerdo con las peticiones del usuario.

Las implementaciones de los sistemas de información y de gestión de recursos basadas en los protocolos de Internet son actualmente la forma más habitual. La interfaz de usuario en estos sistemas se implementa usando un navegador web. La organización de los servidores en estos sistemas refleja el modelo genérico de cuatro capas presentado en la Figura 13.6. Estos sistemas se implementan normalmente como arquitecturas cliente-servidor multicapa, tal y como se indicó en el Capítulo 12. La organización del sistema se muestra en la Figura 13.9. El servidor web es responsable de todas las comunicaciones con el usuario; el servidor de aplicaciones es responsable de implementar la lógica específica de la aplicación, así como las peticiones de almacenamiento y recuperación de información; el servidor de base de datos mueve la información a y desde la base de datos. El uso de múltiples servidores permite un elevado rendimiento y hace posible manejar cientos de transacciones por minuto.

Los sistemas de comercio electrónico son sistemas de gestión de recursos basados en Internet que son diseñados para aceptar peticiones electrónicas de artículos o servicios y a continuación organizar la entrega de estos productos o servicios al cliente. Hay un amplio rango de estos sistemas actualmente en uso que van desde sistemas que permiten servicios tales como la concertación del alquiler de vehículos hasta sistemas que soportan la petición de artículos tangibles tales como libros o comestibles. En un sistema de comercio electrónico, la capa específica de la aplicación incluye funcionalidad adicional para soportar una «venta a la carta» en la que los usuarios pueden incluir varios elementos en distintas transacciones, y a continuación pagar por ellos conjuntamente en una única transacción.

13.3 Sistemas de procesamiento de eventos

Los sistemas de procesamiento de eventos responden a eventos en el entorno del sistema o interfaz del usuario. Tal y como vimos en el Capítulo 11, la principal característica de los sis-

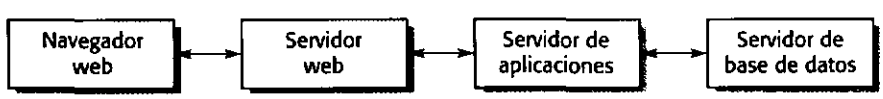


Figura 13.9 Un sistema multicapa de procesamiento de transacciones por Internet.

temas de procesamiento de eventos es que la llegada de los eventos es impredecible y el sistema debe ser capaz de tratar estos eventos cuando ocurran.

Todos nosotros usamos sistemas basados en eventos como éstos en nuestras propias computadoras —procesadores de texto, sistemas de presentaciones y juegos están todos conducidos por eventos desde la interfaz de usuario—. El sistema detecta e interpreta los eventos. Los eventos de interfaz de usuario representan comandos implícitos al sistema, que realiza alguna acción como respuesta a ese comando. Por ejemplo, si usted está utilizando un procesador de texto y pulsa dos veces el ratón sobre una palabra, el evento de doble pulsación significa «selecciona esa palabra».

Los sistemas de tiempo real, los cuales realizan acciones en «tiempo real» como respuesta a algunos estímulos externos, son también sistemas de procesamiento basados en eventos. Sin embargo, para sistemas de tiempo real, los eventos no son normalmente eventos de interfaz de usuario, sino eventos asociados con servidores o actuadores en el sistema. Debido a la necesidad de respuesta en tiempo real a eventos no predecibles, estos sistemas de tiempo real se organizan normalmente como un conjunto de procesos cooperativos. En el Capítulo 15 se tratan las arquitecturas genéricas para sistemas de tiempo real.

En esta sección, nos centramos en la descripción de la arquitectura genérica de los sistemas de edición. Los sistemas de edición son programas que se ejecutan sobre PCs o estaciones de trabajo y que permiten a los usuarios editar documentos tales como documentos de texto, diagramas o imágenes. Algunos editores se centran en la edición de un único tipo de documento, tales como imágenes de una cámara digital o escáner. Otros, entre ellos la mayoría de procesadores de texto, son multieditores e incluyen soporte para editar diferentes tipos de documentos que incluyen texto y diagramas. Incluso se puede pensar en una hoja de cálculo como un sistema de edición en el que se editan las celdas de la hoja. Por supuesto, las hojas de cálculo tienen funcionalidades adicionales para llevar a cabo los cálculos.

Los sistemas de edición tienen varias características que los distinguen de otros tipos sistemas y que influyen en su diseño arquitectónico:

1. Los sistemas de edición son principalmente sistemas para un único usuario. Por lo tanto, no tienen que tratar los problemas de múltiples accesos concurrentes a los datos y tienen una gestión de los datos más sencilla que los sistemas basados en transacciones. Incluso aunque los datos sean compartidos, la gestión de transacciones no se usa normalmente debido a que las transacciones consumen mucho tiempo y se utilizan métodos alternativos para mantener la integridad de los datos.
2. Tienen que proporcionar una rápida realimentación de las acciones del usuario tales como «seleccionar» y «borrar». Esto significa que tienen que funcionar con representaciones de los datos que se almacenan en la memoria de la computadora en vez de en el disco. Debido a que los datos están en la memoria volátil, pueden perderse si hay un fallo en el sistema; por lo tanto, los sistemas de edición deberían realizar alguna previsión para la recuperación de errores.
3. Las sesiones de edición son normalmente mucho más largas que las sesiones que implican peticiones de artículos, o la realización de alguna otra transacción. De nuevo, esto significa que hay un gran riesgo de pérdida si ocurren problemas. Por lo tanto, muchos sistemas de edición incluyen facilidades de recuperación que automáticamente guardan el trabajo en curso y recuperan el trabajo para el usuario en el caso de que se produzca un fallo en el sistema.

Una arquitectura genérica para un sistema de edición se muestra en la Figura 13.10 como un conjunto de objetos que interactúan. Los objetos en el sistema son activos en vez de pasi-

vos (véase el Capítulo 14) y pueden operar concurrentemente y de forma autónoma. Básicamente, los eventos de la pantalla son procesados e interpretados como comandos. Esto actualiza una estructura de datos, que se vuelve a visualizar a continuación sobre la pantalla.

Las responsabilidades de los componentes arquitectónicos mostrados en la Figura 13.10 son:

1. **Pantalla**. Este objeto monitoriza el segmento de memoria de la pantalla y detecta la ocurrencia de eventos. A continuación, estos eventos se envían al objeto de procesamiento de eventos junto con sus coordenadas de pantalla.
2. **Evento**. Este objeto es disparado por un evento originado desde la **Pantalla**. Utiliza el conocimiento sobre lo que se está visualizando para interpretar este evento y traducirlo al comando de edición adecuado. A continuación, este comando se envía al objeto responsable de la interpretación de comandos. Para los eventos más frecuentes, tales como pulsaciones de ratón o pulsaciones de teclas, el objeto evento puede comunicarse directamente con la estructura de datos. Esto permite actualizaciones más rápidas de dicha estructura.
3. **Orden**. Este objeto procesa un comando que proviene del objeto evento y realiza una llamada al método adecuado en el objeto **Editor de datos** para ejecutar el comando.
4. **Editor de datos**. Cuando se llama al comando adecuado en el objeto Editor de datos, se actualiza la estructura de datos y se llama al método **Actualizar** en **Visualización** para visualizar los datos modificados.

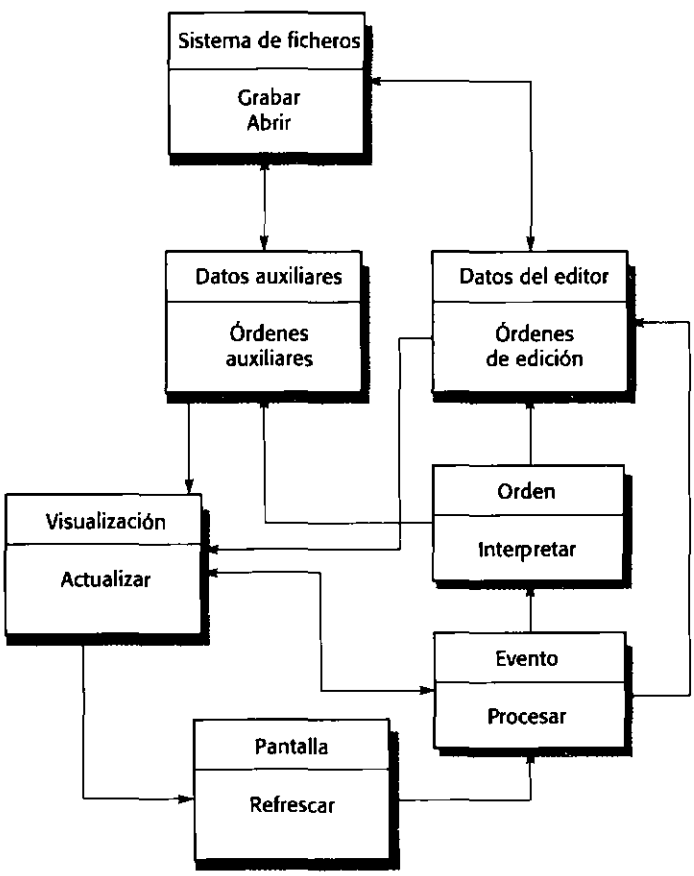


Figura 13.10
Un modelo arquitectónico de un sistema de edición.

5. **Datos auxiliares.** Además de la propia estructura de datos, los editores gestionan otros datos tales como estilos y preferencias. En este modelo arquitectónico sencillo, hemos reunido a todos ellos bajo el nombre de **Datos auxiliares**. Algunos comandos del editor, tales como un comando para iniciar una revisión ortográfica, son implementados por un método de este objeto.
6. **Sistema de ficheros.** Este objeto maneja todas las operaciones para abrir y guardar ficheros. Éstos pueden ser ficheros de datos auxiliares o ficheros del editor de datos. Para evitar la pérdida de datos, muchos editores tienen facilidades de autograbación para guardar la estructura de los datos de forma automática. Ésta puede entonces ser recuperada en caso de ocurrencia del evento de fallo del sistema.
7. **Visualización.** Este objeto mantiene un seguimiento de la organización de lo que se visualiza en la pantalla. Realiza una llamada al método **Refrescar** del objeto **Pantalla** cuando el contenido de la pantalla ha sido modificado.

Debido a la necesidad de una rápida respuesta a los comandos del usuario, los sistemas de edición no tienen un controlador central que realiza llamadas a los componentes para llevar a cabo una acción. En su lugar, los componentes críticos en el sistema se ejecutan concurrentemente y pueden comunicarse directamente (por ejemplo, el procesador de eventos puede comunicarse directamente con el editor de la estructura de datos) para que pueda lograrse un mayor rendimiento.

13.4 Sistemas de procesamiento de lenguajes

Los sistemas de procesamiento de lenguajes aceptan lenguaje natural o artificial como entrada y generan alguna otra representación de ese lenguaje como salida. En ingeniería del software, los sistemas de procesamiento de lenguajes más extensamente usados son los compiladores, que traducen un lenguaje de programación artificial de alto nivel a código máquina, pero otros sistemas de procesamiento de lenguajes traducen una descripción de datos en XML a comandos de consulta de una base de datos y sistemas de procesamiento de lenguaje natural que intentan traducir un lenguaje natural a otro.

En la Figura 13.11 se ilustra la arquitectura de un sistema de procesamiento de lenguajes en su nivel más abstracto. Las instrucciones describen lo que tiene que realizarse y tienen que

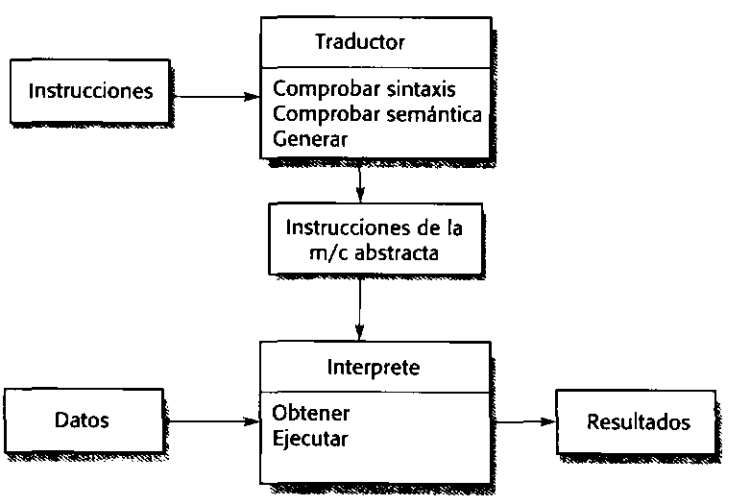


Figura 13.11 La arquitectura abstracta de un sistema de procesamiento de lenguajes.

traducirse por un traductor a algún formato interno. Las instrucciones se corresponden con las instrucciones máquina para una máquina abstracta. A continuación, estas instrucciones son interpretadas por otro componente que obtiene las instrucciones para su ejecución y las ejecuta usando, si es necesario, datos del entorno. La salida del proceso es el resultado de interpretar las instrucciones sobre los datos de entrada. Por supuesto, para muchos compiladores el intérprete es una unidad hardware que procesa instrucciones máquina y la máquina abstracta es un procesador real. Sin embargo, para lenguajes como Java, el intérprete es un componente software.

Los sistemas de procesamiento de lenguajes se usan en situaciones en las que la forma más fácil de resolver un problema es especificar esa solución como un algoritmo o como una descripción de los datos del sistema. Por ejemplo, las herramientas meta-CASE son generadores de programas que se utilizan para crear herramientas CASE específicas para soportar métodos de ingeniería del software. Las herramientas meta-CASE incluyen una descripción de los componentes del método, sus reglas y así sucesivamente, escritas en un lenguaje de propósito específico que es analizada sintáctica y semánticamente para configurar la herramienta CASE generada.

Los traductores en un sistema de procesamiento de lenguajes tienen una arquitectura genérica (Figura 13.12) que incluye los siguientes componentes:

1. Un analizador léxico, que toma como entrada los símbolos del lenguaje y los convierte a un formato interno.
2. Una tabla de símbolos, que almacena información sobre los nombres de las entidades (variables, nombres de clases, nombres de objetos, etc.) usadas en el texto que se está traduciendo.
3. Un analizador sintáctico, que comprueba la sintaxis del lenguaje que se está traduciendo. Utiliza una gramática definida del lenguaje y construye un árbol sintáctico.
4. Un árbol sintáctico, que es una estructura interna que representa al programa que se está compilando.
5. Un analizador semántico, que utiliza información del árbol sintáctico y de la tabla de símbolos para comprobar la corrección semántica del texto en el lenguaje de entrada.
6. Un generador de código, que «recorre» el árbol sintáctico y genera código de máquina abstracta.

También podrían incluirse otros componentes que transforman el árbol sintáctico para mejorar la eficiencia y eliminar redundancias del código máquina generado. En otros tipos de sistemas de procesamiento de lenguajes, tales como un traductor de lenguaje natural, el código generado es realmente el texto de entrada traducido a otro lenguaje.

Los componentes que forman un sistema de procesamiento de lenguajes pueden organizarse de acuerdo con diferentes modelos arquitectónicos. Como apuntan Garlan y Shaw (Garlan y Shaw, 1993), los compiladores pueden implementarse utilizando un modelo compuesto. Puede utilizarse una arquitectura de flujo de datos con la tabla de símbolos actuando como

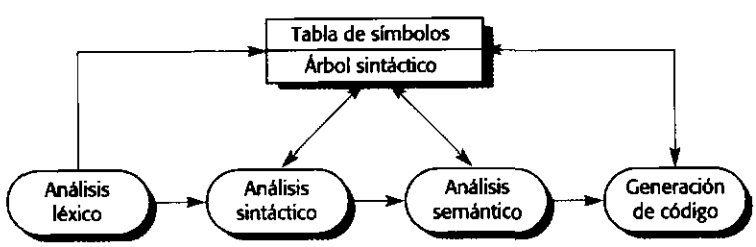


Figura 13.12
Un modelo de flujo de datos de un compilador.

un repositorio para datos compartidos. Las fases de análisis léxico, sintáctico y semántico se organizan de forma secuencial, como se muestra en la Figura 13.12.

Este modelo de compilación de flujo de datos todavía se usa en general. Es efectivo en entornos de procesamiento por lotes en donde los programas son compilados y ejecutados sin la interacción del usuario. Es menos efectivo cuando el compilador tiene que ser integrado con otras herramientas de procesamiento de lenguajes tales como sistemas de edición estructurados, un depurador interactivo o un programa de impresión. A continuación, los componentes genéricos del sistema pueden organizarse en un modelo basado en repositorio, como se muestra en la Figura 13.13.

Esta figura muestra cómo un sistema de procesamiento de lenguajes puede formar parte de un conjunto integrado de herramientas de soporte de programación. En este ejemplo, la tabla de símbolos y el árbol sintáctico actúan como un repositorio central de información. Las herramientas o fragmentos de herramientas se comunican a través de él. Otra información que a menudo está embebida en las herramientas, como la definición de la gramática y la definición del formato de salida del programa, ha sido eliminada de las herramientas y se ha incluido en el repositorio. Por lo tanto, un editor dirigido por la sintaxis puede comprobar que la sintaxis de un programa es correcta al mismo tiempo que está siendo escrito, y una impresora puede generar listados del programa en un formato que es fácil de leer.

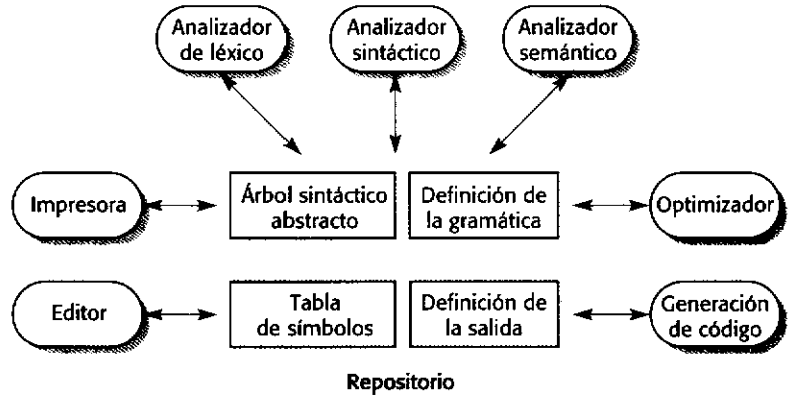


Figura 13.13 El modelo de repositorio de un sistema de procesamiento de lenguajes.

PUNTOS CLAVE

- **Los modelos genéricos de las arquitecturas de sistemas de aplicaciones nos ayudan a entender el funcionamiento de las aplicaciones, comparar aplicaciones del mismo tipo, validar los diseños de los sistemas de aplicaciones y evaluar componentes a gran escala para su reutilización.**
- **Muchas aplicaciones pertenecen a una de las cuatro clases de aplicaciones genéricas o son combinaciones de estas aplicaciones genéricas. Los cuatro tipos de aplicaciones genéricas explicadas aquí son sistemas de procesamiento de datos, sistemas de procesamiento de transacciones, sistemas de procesamiento de eventos y sistemas de procesamiento de lenguajes.**

- **Los sistemas de procesamiento de datos operan en modo diferido y generalmente tienen una estructura entrada-proceso-salida. Los registros se introducen en el sistema, la información se procesa y se generan las salidas.**
- **Los sistemas de procesamiento de transacciones son sistemas interactivos que permiten acceder y modificar la información de una base de datos de forma remota por varios usuarios. Los sistemas de información y los sistemas de gestión de recursos son ejemplos de sistemas de procesamiento de transacciones.**
- **Los sistemas de procesamiento de eventos incluyen a los sistemas de edición y a los sistemas de tiempo real. En un sistema de edición, se interpretan los eventos de la interfaz de usuario y se modifica una estructura de datos de entrada. Los procesadores de texto y los sistemas de presentaciones son ejemplos de sistemas de edición.**
- **Los sistemas de procesamiento de lenguajes se utilizan para traducir textos de un lenguaje a otro y para llevar a cabo las instrucciones especificadas en los lenguajes de entrada. Éstos incluyen un traductor y una máquina abstracta que ejecuta el lenguaje generado.**

LECTURAS ADICIONALES

El tema de las arquitecturas de aplicaciones se ha descuidado durante mucho tiempo; los autores de libros y artículos sobre arquitecturas de software tienden a centrarse en principios abstractos o en arquitecturas de líneas de productos.

Databases and Transaction Processing: An Application-oriented Approach. Éste no es un libro sobre arquitecturas software, pero describe los principios de las aplicaciones de procesamiento de transacciones y centradas en los datos. (P. M. Lewis *et al.*, 2003, Addison-Wesley.)

Design and Use of Software Architectures. Este libro adopta una aproximación de líneas de productos para las arquitecturas software y, por lo tanto, estudia la arquitectura desde la perspectiva de una aplicación. (J. Bosch, 2000, Addison-Wesley.)

EJERCICIOS

13.1 Explique cómo pueden utilizarse las arquitecturas de las aplicaciones genéricas aquí descritas para ayudar al diseñador a tomar decisiones sobre la reutilización del software.

13.2 Usando los cuatro tipos de aplicaciones básicos introducidos en este capítulo, clasifique los siguientes sistemas y explique su clasificación:

- Un sistema de punto de venta en un supermercado.
- Un sistema que envía recordatorios de que deben pagarse las suscripciones a revistas.
- Un sistema de álbum de fotos que proporciona algunas facilidades para restaurar fotografías antiguas.

- Un sistema que lee páginas web para usuarios invidentes.
- Un juego interactivo en el que los personajes se mueven por la pantalla, superan obstáculos y encuentran tesoros.
- Un sistema de control de inventario que mantiene un seguimiento de qué artículos se encuentran en almacén y automáticamente genera órdenes para nuevos pedidos cuando el nivel de almacenamiento está por debajo de un cierto valor.

13.3 Basándose en un modelo de entrada-proceso-salida, amplíe la función Calcular salario de la Figura 13.2 y dibuje un diagrama de flujo de datos que muestre los cálculos llevados a cabo en dicha función. Necesita la siguiente información para realizar esto:

- El registro del empleado identifica la categoría de un empleado. A continuación, esta categoría se usa para buscar en la tabla de salarios.
- A los empleados por debajo de una determinada categoría se les puede pagar las horas extras al mismo precio que las horas de trabajo normales. Las horas extras que se les deben pagar se indican en su registro de empleado.
- La cantidad de impuestos deducidos depende del código de impuestos del empleado (indicado en el registro) y de su salario anual. Las deducciones mensuales para cada código y salario estándar se indican en las tablas de impuestos. Éstas son ampliadas o reducidas proporcionalmente dependiendo de las relaciones entre el salario actual y el salario estándar utilizado.

13.4 Explique por qué la gestión de transacciones en sistemas en los que las entradas del usuario pueden provocar cambios en la base de datos.

13.5 Utilizando el modelo básico de un sistema de información como el presentado en la Figura 13.6, muestre los componentes de un sistema de información que permita a los usuarios ver la información sobre los vuelos de llegada y de salida de un determinado aeropuerto.

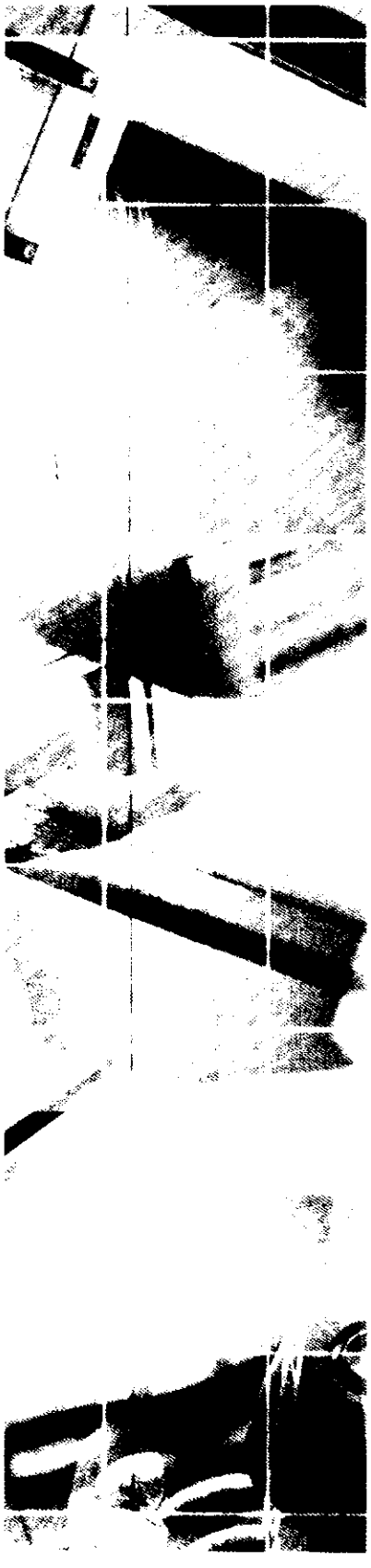
13.6 Utilizando la arquitectura por capas de la Figura 13.8, muestre los componentes de un sistema de gestión de recursos que podría utilizarse para gestionar la reservas de habitaciones de un hotel.

13.7 En un sistema de edición, todos los eventos de la interfaz de usuario pueden ser traducidos en comandos implícitos o explícitos. Explique por qué, en la Figura 13.10, el objeto Evento se comunica directamente con la estructura de datos del editor así como con el objeto Comando.

13.8 Modifique la Figura 13.10 para mostrar la arquitectura genérica de un sistema de hoja de cálculo. Base su diseño en las características de cualquier sistema de hoja de cálculo que usted haya usado.

13.9 ¿Cuál es la función del componente árbol sintáctico en un sistema de procesamiento de lenguajes?

13.10 Usando el modelo genérico de un sistema de procesamiento de lenguajes aquí presentado, diseñe la arquitectura de un sistema que acepte comandos en lenguaje natural y los traduzca en consultas a una base de datos en un lenguaje como SQL.



14

Diseño orientado a objetos

Objetivos

El objetivo de este capítulo es introducir un enfoque de diseño de software en el que el diseño se representa como objetos que interactúan. Cuando termine de leer este capítulo:

- conocerá cómo se representa un diseño de software como un conjunto de objetos que interactúan entre sí y que administran su propio estado y operaciones;
- conocerá las actividades más importantes en un proceso general de diseño orientado a objetos;
- comprenderá los diversos modelos que se utilizan para documentar diseño orientado a objetos;
- habrá sido introducido en la representación de estos modelos en el Lenguaje Unificado de Modelado (UML).

Contenidos

Un sistema orientado a objetos está compuesto de objetos que interactúan, los cuales mantienen ellos mismos su estado local y proveen operaciones sobre su estado (Figura 14.1). La representación del estado es privada y no se puede acceder a ella directamente desde fuera del objeto. El proceso de diseño orientado a objetos comprende el diseño de clases de objetos y las relaciones entre estas clases. Las clases definen los objetos del sistema y sus interacciones. Cuando el diseño se implementa como un programa ejecutable, los objetos requeridos se crean dinámicamente utilizando las definiciones de las clases.

El diseño orientado a objetos es parte del desarrollo orientado a objetos en el que se utiliza una estrategia orientada a objetos en el proceso de desarrollo:

- *El análisis orientado a objetos* comprende el desarrollo de un modelo orientado a objetos del dominio de aplicación. Los objetos identificados reflejan las entidades y operaciones que se asocian con el problema a resolver.
- *El diseño orientado a objetos* comprende el desarrollo de un modelo orientado a objetos de un sistema software para implementar los requerimientos identificados. Los objetos en un diseño orientado a objetos están relacionados con la solución del problema por resolver. Pueden existir relaciones estrechas entre algunos objetos del problema y algunos objetos de la solución, pero inevitablemente el diseñador tiene que agregar nuevos objetos para transformar los objetos del problema e implementar la solución.
- *La programación orientada a objetos* se refiere a implementar el diseño de software utilizando un lenguaje de programación orientado a objetos, como Java. Un lenguaje orientado a objetos provee los recursos para definir las clases y un sistema para crear los objetos correspondientes a las clases.

La transición entre estas etapas de desarrollo se lleva a cabo, idealmente, sin problemas, utilizando notaciones compatibles entre las etapas. Pasar a la siguiente etapa implica refinar la etapa previa agregando algún detalle a las clases existentes y crear nuevas clases con el fin de proveer nuevas funcionalidades. Puesto que la información se oculta dentro de los objetos, las decisiones del diseño detallado de la representación de los datos se puede retrasar hasta que el sistema se implemente. En algunos casos, las decisiones sobre la distribución de los objetos y si éstos se implementan de forma secuencial o concurrente también se pueden retrasar.

Esto significa que los diseñadores de software no están condicionados por los detalles de la implementación del sistema. Pueden formular diseños que se adapten a los diversos entornos de ejecución. Esto es ejemplificado en el enfoque de Arquitectura Dirigida por el Modelo (MDA), el cual propone que el sistema debe ser diseñado explícitamente en dos niveles (Kleppe *et al.*, 2003), un nivel independiente de la implementación y otro dependiente de ésta. Se diseña un modelo abstracto del sistema en el nivel independiente de la implementación, y éste es dirigido hacia un modelo más detallado dependiente de la plataforma, el cual puede utilizarse como base para la generación de código. Actualmente, la aproximación MDA es todavía experimental y no está claro su nivel de adopción.

Los sistemas orientados a objetos son más fáciles de mantener que los sistemas desarrollados con otras aproximaciones, debido a que los objetos son independientes. Tales sistemas pueden ser entendidos y modificados como entidades independientes. Cambiar la implementación de un objeto o agregarle servicios no debe afectar a los otros objetos del sistema. Puesto que los objetos están asociados a cosas, a menudo existe una correspondencia clara entre las entidades del mundo real (como los componentes de hardware) y los objetos de control del sistema. Esto mejora la comprensión y, por lo tanto, la mantenibilidad del diseño.

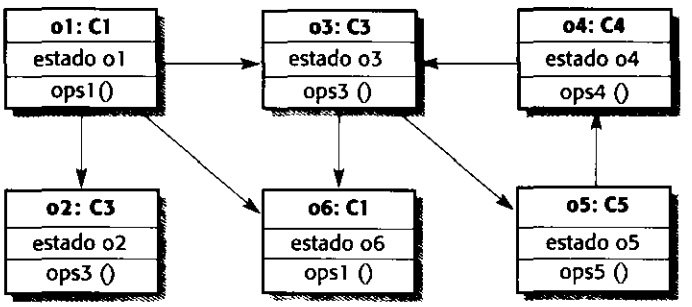


Figura 14.1 Un Sistema compuesto de objetos que interactúan entre sí.

Los objetos son componentes potencialmente reutilizables debido a que son encapsulamientos independientes del estado y las operaciones. Los diseños se pueden desarrollar utilizando objetos creados en los diseños previos. Esto reduce los costes de diseño, programación y validación. También conduce a la utilización de objetos estándar (por lo que se mejora la comprensión del diseño) y reduce los riesgos implicados en el desarrollo de software. Sin embargo, como se explica en los Capítulos 18 y 19, algunas veces la reutilización se implementa de una mejor forma si se utilizan colecciones de objetos (componentes o marcos de trabajo) en lugar de objetos individuales.

Se han propuesto diversos métodos de diseño orientado a objetos (Coad y Yourdon, 1990; Robinson, 1992; Jacobson *et al.*, 1993; Graham, 1994; Booch, 1994). Se ha definido una unificación de las notaciones utilizadas en estos métodos (UML). El Proceso Unificado de Racional (RUP), que se analizó en el Capítulo 4, ha sido diseñado para explotar los modelos que pueden expresarse en UML (Rumbaugh *et al.*, 1999). La notación UML se utilizará a lo largo de este capítulo.

En el Capítulo 17 se expondrá un sistema desarrollado basado en un diseño amplio que puede ser criticado debido al extenso esfuerzo de análisis y diseño no ajustado al desarrollo y entrega incremental. Para atajar este problema, se han desarrollado los llamados métodos ágiles, los cuales reducen o eliminan completamente la actividad de diseño orientado a objetos. Nuestro punto de vista respecto a este tema es amplio, el diseño «pesado» no es necesario en sistemas de negocios de tamaño pequeño o mediano. Sin embargo, para sistemas grandes, particularmente en sistemas críticos, es esencial asegurar que los equipos trabajen en diferentes partes del sistema adecuadamente coordinados. Por esta razón, no se han usado los ejemplos previos de la biblioteca o del sistema de inyección de insulina en este capítulo. En su lugar, se ha utilizado un ejemplo que es parte de un sistema mucho mayor donde el enfoque del diseño orientado a objetos es más útil.

Esta visión es reflejada, y en algunos casos extendida, en el Proceso Unificado de Racional que orienta el desarrollo iterativo y entregas incrementales de grandes sistemas software. Este proceso es un proceso de desarrollo iterativo basado alrededor de casos de uso para expresar los requerimientos y el diseño orientado a objetos, centrándose particularmente en el diseño de la arquitectura.

El proceso de diseño que se describe en la Sección 14.2 tiene algunas cosas en común con el RUP, pero con menos énfasis en el desarrollo dirigido por casos de uso. La utilización de casos de uso significa que el diseño está ciertamente centrado en el usuario y basado alrededor de las interacciones del usuario con el sistema. Sin embargo, representar los requerimientos que no están directamente ligados a los usuarios del sistema mediante casos de uso es difícil. Los casos de usos tienen ciertamente un papel en el análisis y diseño orientado a objetos, pero necesitan ser complementados con otras técnicas que nos permitan descubrir requerimientos indirectos y no funcionales del sistema.

14.1 Objetos y clases

En la actualidad los términos «objeto» y «orientado a objetos» se aplican a diversos tipos de entidades, métodos de diseño, sistemas y lenguajes de programación. Sin embargo, por lo general se acepta que un objeto es un encapsulamiento de la información, y esto se refleja en las siguientes definiciones de objeto y clase:

> *Un objeto es una entidad que tiene un estado y un conjunto de operaciones definidas que operan sobre ese estado. El estado se representa como un conjunto de atributos del objeto. Las operaciones asociadas al objeto proveen servicios a otros objetos (clientes) que solicitan estos servicios cuando se requiere llevar a cabo algún cálculo.*
>
> *Los objetos se crean conforme a una definición de clases de objetos. Una definición de clases sirve como una plantilla para crear objetos. Ésta incluye las declaraciones de todos los atributos y operaciones asociados con un objeto de esa clase.*

En UML, una clase se representa como un rectángulo al cual se le han asignado un nombre y dos secciones. Los atributos del objeto se listan en la sección superior. Las operaciones que están asociadas con el objeto se listan en la sección inferior. La Figura 14.2 ilustra esta notación par utilizando una clase de objetos que modela a un empleado de una organización. En la notación UML, el término «operación» es la especificación de una acción: el término «método» se utiliza para referirse a la implementación de la operación.

La clase **Employee** define varios de los atributos que contienen información de los empleados, incluyendo su nombre y dirección, el número de seguro social, el código de pago de impuestos, etcétera. Los puntos suspensivos (...) indican que existen más atributos asociados con la clase y que no se muestran aquí. Las operaciones asociadas con el objeto son **join** (que se llama cuando un empleado se une a la organización), **leave** (que se llama cuando un empleado abandona la organización), **retire** (que se llama cuando un empleado se convierte en pensionista) y **changeDetails** (que se llama cuando la información de un empleado requiere algunos cambios).

Los objetos se comunican a través de la solicitud de servicios (llamando a los métodos) de otros objetos y, si es necesario, intercambiando la información requerida para que ese servi-

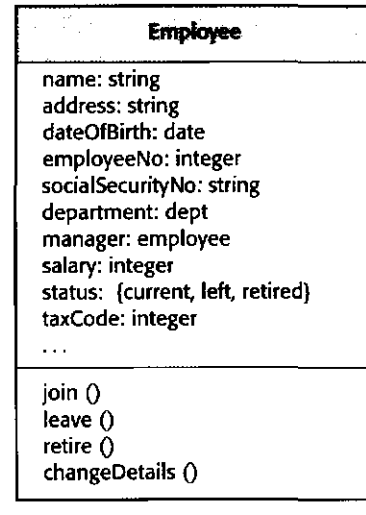


Figura 14.2 Un objeto empleado.

cio se suministre. Las copias de la información necesaria para ejecutar el servicio y los resultados de la ejecución del servicio se pasan como parámetros. Algunos ejemplos de este estilo de comunicación son:

```
// Llama a un método asociado con un objeto búfer
// que regresa el siguiente valor en el búfer
v = circula rBuffer.Get ();
// Llama al método asociado con un objeto
// termostato que establece la temperatura que se mantendrá
thermostat.setTemp (20);
```

En los sistemas basados en servicios las comunicaciones de los objetos se implementan directamente como mensajes de texto que los objetos intercambian. El objeto receptor analiza el mensaje, identifica el servicio y los datos asociados, y lleva a cabo el servicio solicitado. Sin embargo, cuando los objetos coexisten en el mismo programa, las llamadas a los métodos se implementan como llamadas a procedimientos o funciones en un lenguaje como C.

Cuando las peticiones de servicios son implementadas siguiendo esta vía, la comunicación entre los objetos es síncrona. Esto es, el objeto invocador espera a que la petición del servicio se complete. Sin embargo, si los objetos se implementan como procesos concurrentes o hilos, la comunicación es asíncrona. El objeto invocador continúa en operación mientras que el servicio solicitado se ejecuta. Más adelante en esta sección, se explica cómo se implementan los objetos como procesos concurrentes.

Como se indicó en el Capítulo 8, en la parte donde se describen los diferentes modelos de objetos posibles, las clases se pueden organizar mediante una generalización o jerarquía de herencia que muestra las relaciones entre las clases generales y más específicas. La clase más específica concuerda completamente con la clase general, pero incluye información adicional. En la notación UML, la generalización se indica mediante una flecha que apunta a la clase padre. En los lenguajes de programación orientados a objetos, por lo regular la generalización se implementa utilizando el mecanismo de herencia. La clase hija hereda los atributos y operaciones de la clase padre.

La Figura 14.3 muestra un ejemplo de jerarquía de clases en el cual se ilustran las diversas clases para **Employee**. Las clases inferiores de la jerarquía tienen los mismos atributos y operaciones que sus clases padre, y se les pueden agregar nuevos atributos y operaciones o modificar los de sus clases padre. Esto significa que el intercambio sólo existe en un solo sentido. Si el nombre de la clase padre se utiliza en un modelo, esto implica que el objeto en el sistema se define ya sea como esa clase o como cualquiera de sus descendientes.

La clase **Manager** de la Figura 14.3 tiene todos los atributos y operaciones de la clase **Employee**, pero además tiene dos nuevos atributos que registran los presupuestos controlados por el administrador y la fecha en la que el administrador fue solicitado para una tarea de administración particular. De igual forma, la clase **Programmer** contiene nuevos atributos que definen el proyecto en el que trabaja el programador y las capacidades que tiene. Los objetos de la clase **Manager** o **Programmer** se pueden utilizar en cualquier lugar en el que se requiera un objeto de la clase **Employee**.

Los objetos que son miembros de una clase participan en las relaciones con otros objetos. Estas relaciones se modelan describiendo las asociaciones entre las clases. En UML, las asociaciones se denotan mediante una línea que une las clases, a la que se le puede agregar una

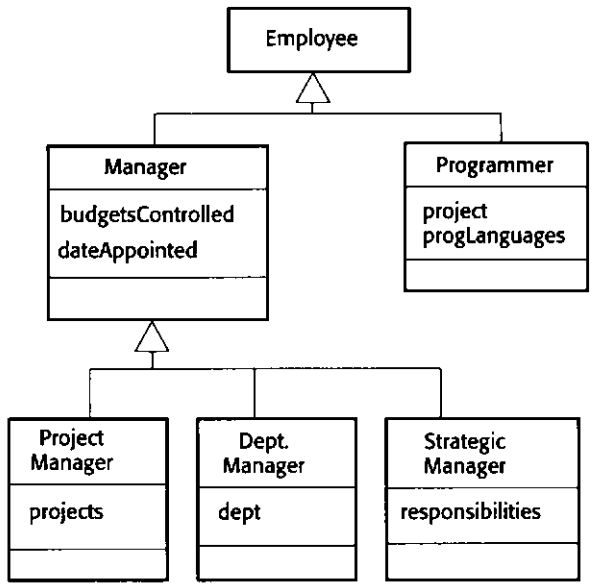


Figura 14.3 Un ejemplo de jerarquía con generalización.

nota con información de la asociación. Esto se ilustra en la Figura 14.4, que muestra la asociación entre los objetos de la clase **Employee** y los objetos de la clase **Department** y entre los objetos de la clase **Employee** y los de la clase **Manager**.

La asociación es una relación muy general y a menudo se utiliza en UML para indicar que un atributo del objeto es un objeto asociado, o que la implementación de un método del objeto depende del objeto asociado. Sin embargo, al menos en principio, cualquier clase de asociación es posible. Una de las asociaciones más comunes es la agregación que muestra la manera en que los objetos están compuestos de otros objetos. Véase el Capítulo 8 para una descripción de este tipo de asociación.

14.1.1 Objetos concurrentes

De forma conceptual, un objeto solicita un servicio de otro objeto enviándole un mensaje de «solicitud de servicio» a ese objeto. No es necesario que un objeto ejecute estos mensajes en forma secuencial y que espere hasta completar un servicio solicitado. En consecuencia, el modelo general de interacción de objetos les permite ejecutarse concurrentemente como procesos en paralelo. Estos objetos se ejecutan en la misma computadora o como objetos distribuidos en diferentes máquinas.

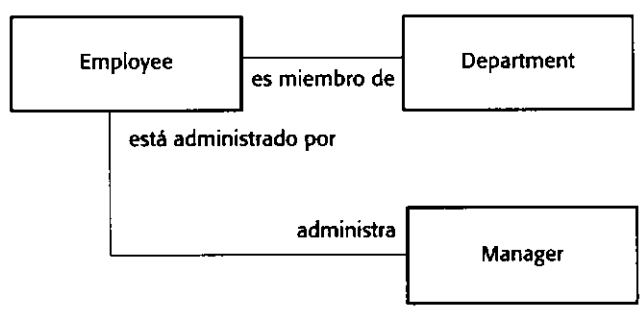


Figura 14.4 Un modelo de asociación.

En la práctica, muchos de los lenguajes de programación orientados a objetos tienen su propio modelo de ejecución en serie en el cual las peticiones de los servicios a objetos se implementan de la misma forma en la que se llama a una función. Por lo tanto, cuando un objeto llamado **theList** se crea a partir de una clase de objetos normal, en Java escribimos:

```
theList.append (17)
```

Éste llama al método **append** asociado con el objeto **theList** para añadir el elemento 17 a **theList**, y la ejecución del objeto que hace la llamada se suspende hasta que la operación **append** se ha completado. Sin embargo, Java incluye un mecanismo muy simple (hilos) que nos permite crear objetos que se ejecuten al mismo tiempo. Los hilos se crean en Java utilizando la clase nativa **Threat** como clase padre en la declaración de una clase. Los hilos deben incluir un método llamado **run**, el cual es inicializado por el run-time de Java cuando se crean los objetos definidos como hilos. Por lo tanto, es fácil tomar un diseño orientado a objeto e implementarlo de forma que los objetos sean procesos concurrentes.

Existen dos clases de implementación de objetos concurrentes:

1. Servidores en los cuales el objeto se implementa como un proceso paralelo con métodos que corresponden a las operaciones definidas de los objetos. Los métodos inician su actividad en respuesta a un mensaje externo y se ejecutan en paralelo con los métodos asociados a otros objetos. Cuando han completado su operación, el objeto se suspende por sí mismo y espera por las peticiones adicionales de los servicios.
2. Objetos activos en los cuales el estado de los objetos cambia debido a la ejecución de operaciones internas. El proceso que representa al objeto ejecuta continuamente estas operaciones, por lo que no se suspende por sí solo.

Los servidores son más útiles en entornos distribuidos donde los objetos invocadores y los objetos invocados se ejecutan en diferentes computadoras. El tiempo de respuesta para el servicio solicitado es impredecible; por lo tanto, siempre que sea posible, se debe diseñar el sistema de tal forma que el objeto que solicita un servicio no tenga que esperar a que el servicio se complete. También se pueden utilizar en una sola máquina en la cual a un servicio le lleve cierto tiempo completarse (por ejemplo, la impresión de un documento) y el servicio puede ser requerido por varios objetos diferentes.

Los objetos activos se utilizan cuando un objeto necesita actualizar su propio estado en intervalos específicos. Esto es común en sistemas de tiempo real donde los objetos se asocian a dispositivos de hardware que recolectan información del entorno del sistema. Los métodos del objeto permiten a otros objetos acceder a la información del estado.

La Figura 14.5 muestra la manera en que un objeto activo se define e implementa en Java. Esta clase de objetos representa un transpondedor de un avión. Dicho transpondedor mantiene información de la posición del avión utilizando un sistema de navegación vía satélite. Puede responder a los mensajes de las computadoras para el control del tráfico aéreo. Suministra la posición actual del avión como respuesta a la petición del método **givePosition**. Este objeto se implementa como un hilo, donde un ciclo continuo en el método **run** incluye código para calcular la posición del avión utilizando señales de los satélites.

```
class Transponder extends Thread {

    Position currentPosition ;
    Coords c1, c2 ;
    Satellite sat1, sat2 ;
    Navigator theNavigator ;

    public Position givePosition ()
    {
        return currentPosition ;
    }

    public void run ()
    {
        while (true)
        {
            c1 = sat1.position () ;
            c2 = sat2.position () ;
            currentPosition = theNavigator.compute (c1, c2) ;
        }
    }
} //Transponder
```

Figura 14.5 Implementación de un objeto activo utilizando subprocesos de Java.

14.2 Un proceso de diseño orientado a objetos

En esta sección se ilustra el proceso de diseño orientado a objetos por medio del desarrollo de un ejemplo de diseño para el software de control que está incrustado en una estación meteorológica automatizada. Como se indicó en la introducción, existen varios métodos de diseño orientados a objetos sin que exista un «mejor» método o proceso de diseño. El proceso que aquí se muestra es un proceso general que incorpora las actividades comunes de muchos de los procesos de diseño orientados a objetos.

El proceso general que aquí se utiliza para el diseño orientado a objetos tiene varias etapas:

1. Comprender y definir el contexto y los modos de utilización del sistema.
2. Diseñar la arquitectura del sistema.
3. Identificar los objetos principales en el sistema.
4. Desarrollar los modelos de diseño.
5. Especificar las interfaces de los objetos.

A propósito, este proceso no se muestra como un diagrama de proceso simple, puesto que esto implicaría la existencia de actividades detalladas llevadas a cabo en secuencia. De hecho, todas las actividades anteriores se pueden ver como actividades entrelazadas que influyen entre sí. Los objetos se identifican y las interfaces se especifican completa o parcialmente en el momento de definir la arquitectura del sistema. En el momento de elaborar los modelos de objetos, estas definiciones de los objetos se refinan y pueden implicar un cambio en la arquitectura del sistema.

Más adelante, en esta sección, se expondrá de forma separada cada una de estas etapas del proceso de diseño. Sin embargo, no debe suponerse que el diseño es un proceso sencillo y bien estructurado. En realidad, un diseño se desarrolla proponiendo soluciones y refinando estas soluciones, al disponer de más información. Inevitablemente, se tendrá que regresar y retomar los problemas cuando éstos surjan. Algunas veces, tendrá que explorar las opciones en detalle para ver si funcionan: otras veces habrá que prescindir de los detalles y retomarlos más adelante era el proceso.

Estas actividades del proceso se ilustran mediante un ejemplo de un diseño orientado a objetos. Este ejemplo es parte de un sistema para trazar mapas meteorológicos, utilizando datos meteorológicos recogidos de forma automática. El detalle de los requerimientos para este sistema abarcaría varias páginas. Sin embargo, la arquitectura del sistema completo se puede desarrollar a partir de una descripción relativamente breve del sistema:

Se requiere un sistema para generar mapas meteorológicos a partir de la recogida periódica de los datos de estaciones meteorológicas remotas y automáticas y datos de otras fuentes, como observatorios meteorológicos, globos y satélites. Las estaciones meteorológicas transmiten sus datos a la computadora del área en respuesta a una petición de esa máquina.

El sistema de cómputo del área valida los datos recogidos e integra los datos de diversas fuentes. Los datos integrados se guardan y, utilizando datos de este archivo y una base de datos de mapas digitalizados, se crea un conjunto local de mapas meteorológicos. Los mapas se pueden imprimir para su distribución en una impresora de mapas de propósito especial y pueden ser visualizados en pantalla en diferentes formatos.

Esta descripción muestra que parte del sistema completo se refiere a la recogida de datos, parte a la integración de los datos de diversas fuentes, parte a archivar estos datos y parte a crear mapas meteorológicos. La Figura 14.6 ilustra una posible arquitectura del sistema que se deriva de esta descripción. Ésta es una arquitectura de capas (descrita en el Capítulo 11) que refleja las diversas etapas de procesamiento en el sistema, principalmente recogida de datos, integración de datos, archivado de datos y generación de mapas. En este caso, una arquitectura de capas es apropiada debido a que cada etapa depende sólo del procesamiento de la etapa previa a esta operación.

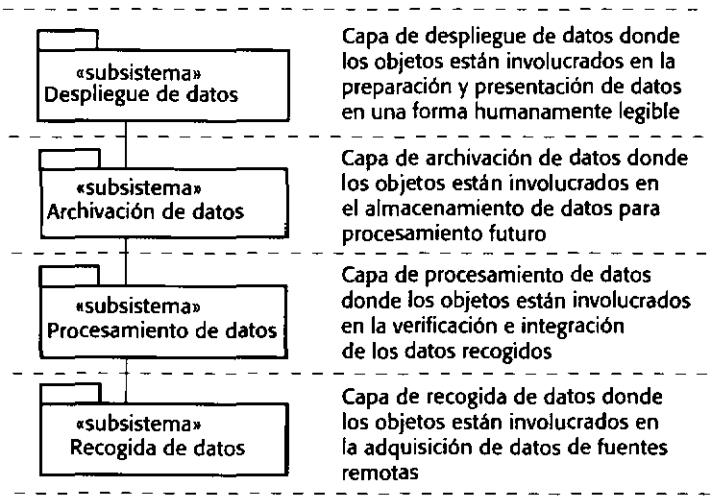


Figura 14.6 Arquitectura de capas para un sistema de creación de mapas meteorológicos.

En la Figura 14.6 se muestran las diferentes capas y se incluye el nombre de la capa en un símbolo de representación de paquetes en UML que se ha denotado como un subsistema. Un paquete en UML representa una colección de objetos y otros paquetes. Aquí se ha utilizado para mostrar que cada capa incluye otros componentes.

En la Figura 14.7 se ha extendido este modelo de arquitectura abstracta para mostrar los componentes de los subsistemas. Éstos siguen siendo muy abstractos y se han derivado de la información en la descripción del sistema. A partir de ahora, el ejemplo de diseño se centra en el subsistema de la estación meteorológica que es parte de la capa de recogida de datos.

14.2.1 Contexto del sistema y modelos de utilización

La primera etapa en el proceso de diseño de software es comprender las relaciones entre el software que se está diseñando y el entorno externo. Comprender esto ayuda a decidir cómo suministrar la funcionalidad requerida al sistema y cómo estructurar éste para que se comunique efectivamente con su entorno.

El contexto del sistema y el modelo de utilización del sistema representan dos modelos complementarios entre un sistema y su entorno:

1. El contexto del sistema es un modelo estático que describe a los otros sistemas de su entorno.
2. El modelo de utilización del sistema es un modelo dinámico que describe cómo el sistema interactúa con su entorno.

El modelo de contexto de un sistema se representa utilizando asociaciones (véase la Figura 14.4) donde, esencialmente, se produce un diagrama de bloques sencillo de la arquitectura del sistema completo.

Esto se puede extender representando un modelo del subsistema utilizando paquetes en UML como se muestra en la Figura 14.7. Esto ilustra que el contexto de la estación meteorológica está dentro de un sistema involucrado en la recolección de datos. También muestra otros subsistemas que componen el sistema de trazado de mapas meteorológicos.

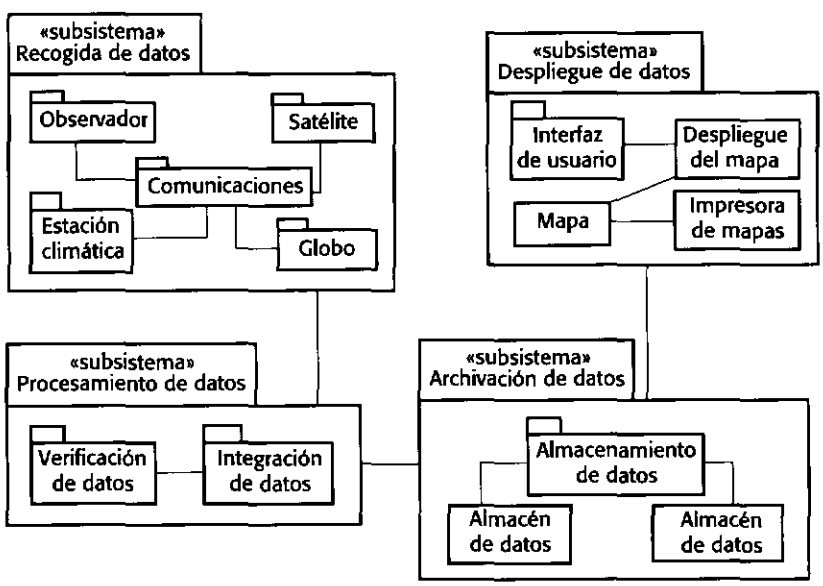


Figura 14.7 Subsistemas en el ejemplo de mapas meteorológicos.

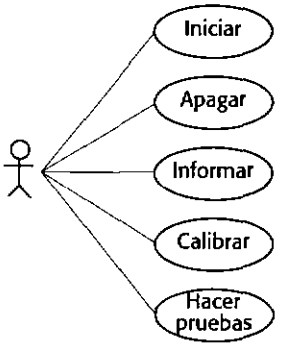


Figura 14.8 Casos de uso de la estación meteorológica.

Cuando se modelan las interacciones de un sistema con su entorno, se debe utilizar un enfoque abstracto que no incluya mucho detalle de esas interacciones. El enfoque que se propone en UML es desarrollar un modelo de casos de uso donde cada caso de uso representa una interacción con el sistema. En los modelos de caso de uso (descritos también en el Capítulo 7), cada interacción posible se enuncia en una elipse y la entidad externa implicada en la interacción se representa mediante una figura estilizada. En el caso del sistema de la estación meteorológica esta entidad externa no es un humano sino el sistema de procesamiento para los datos meteorológicos.

En la Figura 12.8 se presenta un modelo de caso de uso para la estación meteorológica. Éste muestra que la estación meteorológica interactúa con las entidades externas para iniciarse o para apagarse, para informar de los datos meteorológicos recogidos y para calibrar y probar los instrumentos.

Cada uno de estos casos de uso se describe utilizando una descripción en lenguaje natural sencillo. Esto ayuda a los diseñadores a identificar los objetos en el sistema y les permite comprender lo que hará el sistema. Aquí se utiliza un formulario estandarizado para describir e identificar claramente la información que se intercambia, cómo se inicia la interacción, etcétera. Esto se muestra en la Figura 14.9 donde se describe el caso de uso **Informar** de la Figura 14.8.

Sistema	**Estación meteorológica.**
Caso de uso	**Informar.**
Actores	**Sistema de recogida de datos meteorológicos, estación meteorológica.**
Datos	**La estación meteorológica envía un resumen de los datos meteorológicos recogidos de los instrumentos en el periodo de recogida al sistema de recogida de datos meteorológicos. Los datos enviados son los valores máximo, mínimo y promedio de las temperaturas del suelo y del aire; las presiones del aire máxima, mínima y promedio; las velocidades del viento máxima, mínima y promedio; la lluvia total y la dirección del viento tomada cada cinco minutos.**
Estímulo	**El sistema de recogida de datos meteorológicos establece un vínculo por medio del módem con la estación meteorológica y solicita la transmisión de los datos.**
Respuesta	**Un resumen de los datos se envía al sistema de recogida de datos meteorológicos.**
Comentarios	**Habitualmente, a las estaciones meteorológicas se les pide informar una vez por hora, aunque esta frecuencia puede diferir de una estación a otra y puede modificarse en el futuro.**

Figura 14.9 Descripción del caso de uso Informar.

La descripción del caso de uso ayuda a identificar los objetos y operaciones en sistema. De la descripción del caso de uso **Informar**, es obvio que se requieren objetos que representen los instrumentos que recogen los datos meteorológicos, así como el objeto que represente el resumen de dichos datos. Es necesario definir las operaciones que representen la solicitud y el envío de datos meteorológicos.

14.2.2 Diseño de la arquitectura

Una vez que se han definido las interacciones entre el sistema de software que se está diseñando y el entorno del sistema, se puede utilizar esta información como base para diseñar la arquitectura del sistema. Por supuesto, es necesario combinar esto con el conocimiento general de los principios de diseño arquitectónico y con el conocimiento más detallado del dominio.

La estación meteorológica automatizada es un sistema relativamente sencillo y su arquitectura se puede representar como un modelo en capas. Esto se ilustra en la Figura 14.10 como un árbol de paquetes de UML dentro del paquete más general Estación meteorológica. Note que aquí se ha utilizado la notación en UML (texto en cuadros con una pestaña en la esquina) para proveer información adicional.

Las tres capas en el software de la estación meteorológica son:

1. *La capa de interfaz*, que comprende todas las comunicaciones con otras partes del sistema y el suministro de las interfaces externas del sistema.
2. *La capa de recogida de datos*, que se ocupa de administrar la recogida de datos de los instrumentos y de resumir los datos meteorológicos antes de la transmisión al sistema de mapas.
3. *La capa de instrumentos*, que comprende el encapsulamiento de todos los instrumentos utilizados en la recogida de los datos acerca de las condiciones meteorológicas.

En general, se debe tratar de descomponer un sistema de tal forma que las arquitecturas sean lo más sencillas posible. Una regla que siempre funciona es que no debe haber más de siete entidades fundamentales en un modelo arquitectónico. Cada una de estas entidades se puede describir independientemente, pero, por supuesto, se debe dejar al descubierto la estructura de las entidades, como se muestra en la Figura 14.7.

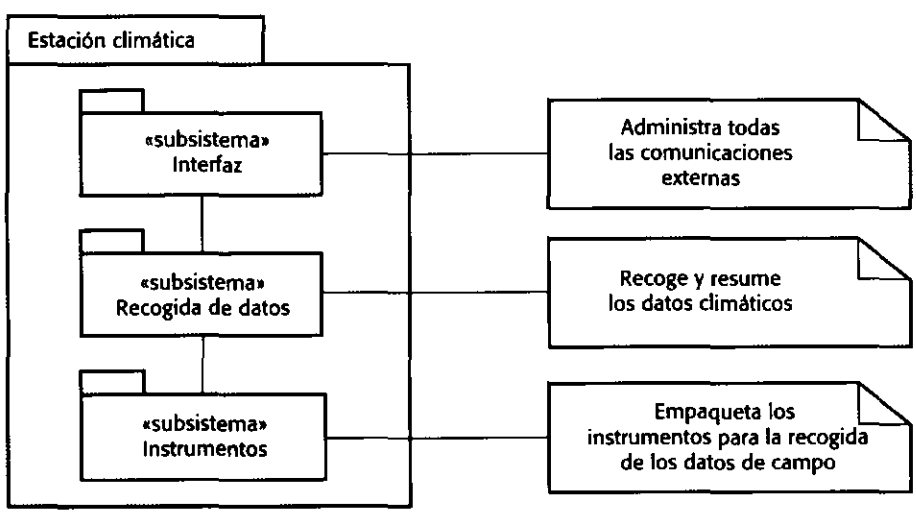


Figura 14.10 La arquitectura de una estación meteorológica.

14.2.3 Identificación de objetos

En esta etapa del proceso de diseño ya tuvo que haber formulado algunas ideas de los objetos esenciales del sistema que se está diseñando. En el sistema de la estación meteorológica, está claro que los instrumentos son objetos y que se necesita al menos un objeto en cada uno de los niveles de la arquitectura. Esto refleja un principio general, el cual señala que los objetos tienden a emerger durante el proceso de diseño. Sin embargo, por lo general, hay que buscar y documentar otros objetos que pudieran ser relevantes.

Aunque esta sección se ha denominado identificación de objetos, en la práctica este proceso se refiere a la identificación de clases. El diseño se describe en función de estas clases. De forma inevitable, se tienen que refinar las clases identificadas de forma inicial y volver a esta etapa del proceso conforme se vaya obteniendo una comprensión más profunda del diseño.

Se han hecho varias propuestas de cómo identificar las clases:

1. Utilizar un análisis gramatical de la descripción en lenguaje natural del sistema. Los objetos y los atributos son sustantivos; las operaciones o servicios son verbos (Abbott, 1983). Este enfoque se consideró en el método HOOD para el diseño orientado a objetos (Robinson, 1992) que se utiliza ampliamente en la industria aeroespacial europea.
2. Utilizar entidades tangibles (cosas) en el dominio de aplicación como aviones, papeles como administrador, eventos como una petición, interacciones como reuniones, ubicaciones como oficinas, unidades organizacionales como compañías, etcétera (Shlaer y Mellor, 1998; Coad y Yourdon, 1990; Wirfs-Brock *et al.*, 1990). Esto se debe complementar identificando estructuras de almacenamiento (estructuras abstractas de datos) en el dominio de la solución, las cuales podrían requerirse para apoyar a estos objetos.
3. Utilizar un enfoque de comportamiento en el cual el diseñador primero comprende el comportamiento total del sistema. Los diversos comportamientos se asignan a distintas partes del sistema para así comprender quién inicia y participa en estos comportamientos. Los participantes que desempeñan papeles importantes se identifican como objetos (Rubin y Goldberg, 1992).
4. Utilizar un análisis basado en escenarios en el cual se identifican y analizan en su momento varios escenarios de la forma de utilizar el sistema. Puesto que cada escenario se analiza, el equipo responsable del análisis debe identificar los objetos, atributos y operaciones requeridos. Para ayudar a este enfoque basado en escenarios existe un método efectivo de análisis denominado tarjetas CRC en el cual los analistas y diseñadores se encargan de identificar las actividades de los objetos (Beck y Cunningham, 1989).

Estos enfoques ayudan en el inicio de la identificación de objetos. En la práctica, diversas fuentes de conocimiento se utilizan para descubrir objetos y clases. Los objetos y las operaciones que se identifican inicialmente de la descripción informal del sistema pueden ser un punto de partida para el diseño. La información adicional del conocimiento del dominio de aplicación o del análisis del escenario se utiliza para refinar y extender los objetos iniciales. Esta información se recoge de los documentos de requerimientos, de las discusiones con los usuarios y de un análisis de los sistemas existentes.

Aquí se ha utilizado un enfoque híbrido para identificar los objetos de la estación meteorológica. No se cuenta con suficiente espacio para describir a todos los objetos; sin embargo, en la Figura 14.11 se muestran cinco clases de objetos: **Ground Thermometer**, **Anemometer** y **Barometer** representan objetos del dominio de la aplicación, y **WeatherStation** y **WeatherData** provienen de la descripción del sistema y de la descripción del escenario (casos de uso).

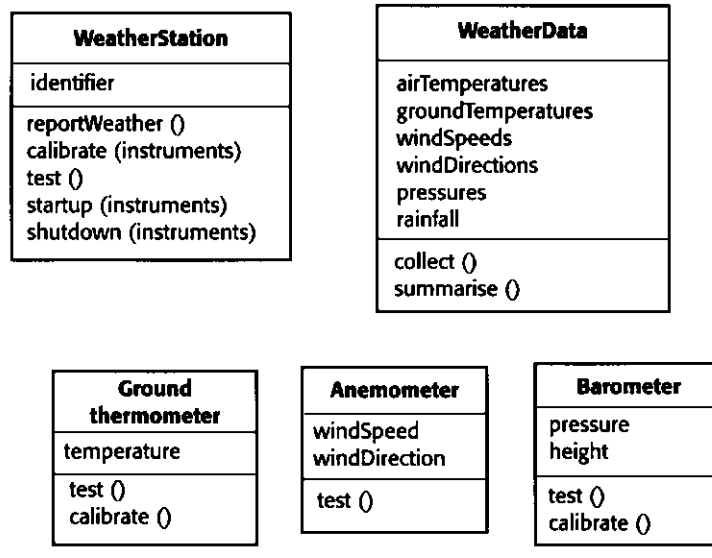


Figura 14.11 Ejemplos de clases en el sistema de la estación meteorológica.

Estos objetos se relacionan con los diversos niveles de la arquitectura del sistema.

1. La clase **WeatherStation** provee la interfaz básica de la estación meteorológica con su entorno. Por lo tanto, sus operaciones reflejan las interacciones que se muestran en la Figura 14.8. En este caso se utiliza una sola clase para encapsular todas estas interacciones, pero en otros diseños puede ser más apropiado utilizar varias clases para proveer la interfaz del sistema.
2. La clase de objetos **WeatherData** encapsula el resumen de datos de los diversos instrumentos en la estación meteorológica. Sus operaciones asociadas se refieren a la recogida y resumen de los datos requeridos.
3. Las clases **GroundThermometer, Anemometer** y **Barometer** se relacionan directamente con los instrumentos del sistema. Reflejan las entidades de hardware tangibles en el sistema y las operaciones se refieren al control de ese hardware.

En esta etapa del proceso de diseño, se utiliza el conocimiento del dominio de la aplicación para identificar a los objetos y servicios adicionales. En este caso, se sabe que a menudo las estaciones climatológicas se localizan en lugares remotos y éstas incluyen diversos instrumentos que algunas veces funcionan mal. Los fallos en los instrumentos deberán informarse de forma automática. Esto implica que son necesarios los atributos y operaciones para verificar el funcionamiento correcto de los instrumentos. Obviamente, existen muchas estaciones climatológicas remotas. Por lo tanto, se necesita una forma de identificar los datos recogidos de cada estación, por lo que cada estación climatológica debe tener su propio identificador.

En este ejemplo, se ha decidido no hacer objetos activos a los objetos asociados con cada instrumento. La operación **collect** en **WeatherData** hace una llamada a los objetos instrumento para realizar las lecturas cuando éstas se requieran. Los sujetos activos incluyen su propio control y, en este caso, esto significa que cada instrumento decide cuándo llevar a cabo las lecturas. Sin embargo, la desventaja de esto es que, si se tomó la decisión de cambiar los tiempos de recogida de datos o si las diversas estaciones climatológicas recogen datos de forma diferente, tienen que introducirse nuevas clases de objetos. Al hacer que los objetos instrumento realicen lecturas bajo petición, cualquier cambio en la estrategia de recogida se puede implementar fácilmente sin cambiar los objetos asociados con los instrumentos.

14.2.4 Modelos de diseño

Los modelos de diseño muestran los objetos o clases en un sistema y, donde sea apropiado, los diferentes tipos de relaciones entre estas entidades. Los modelos de diseño son esencialmente el diseño mismo. Son el puente entre los requerimientos y la implementación del sistema. Esto significa que existen requerimientos en conflicto en estos modelos. Tienen que ser abstractos con el fin de que el detalle innecesario no oculte las relaciones entre ellos y los requerimientos del sistema. Sin embargo, también tienen que incluir suficiente detalle para que los programadores tomen las decisiones de implementación.

En general, le puede dar la vuelta a este conflicto desarrollando diversos modelos en diversos niveles de detalle. Si existen vínculos cercanos entre los ingenieros de requerimientos, los diseñadores y los programadores, lo único que se requiere son modelos abstractos. Las decisiones del diseño específico se construyen durante la implementación del sistema. Si los vínculos entre los especificadores del sistema, los diseñadores y los programadores son indirectos (por ejemplo, cuando un sistema se diseña en una parte de la organización pero se implementa en otra), se requieren modelos más detallados.

Un paso importante en el proceso de diseño es decidir qué modelos de diseño son necesarios y el nivel de detalle de estos modelos. Esto también depende del tipo de sistema que se esté desarrollando. Un sistema de procesamiento secuencial de datos se diseña de forma diferente de un sistema dedicado en tiempo real, por lo que utiliza distintos modelos de diseño. Existen muy pocos sistemas donde todos los modelos son necesarios. Minimizar el número de modelos que se produce reduce los costes de diseño y el tiempo requerido para completar el proceso.

Existen dos tipos de modelos de diseño para describir un diseño orientado a objetos:

1. Modelos estáticos que describen la estructura estática del sistema en términos de las clases del sistema y sus relaciones. Las relaciones importantes que se documentan en esta etapa son de generalización, utiliza/utilizado-por y de composición.
2. Modelos dinámicos que describen la estructura dinámica del sistema y que muestran las interacciones entre los objetos del sistema (no entre las clases). Las interacciones que se documentan incluyen la secuencia de servicios solicitados por los objetos y la forma en que el estado del sistema se relaciona con estas interacciones de objetos.

UML provee 12 modelos estáticos y dinámicos que pueden ser utilizados en el documento de diseño. No se cuenta con suficiente espacio para abordar a todos ellos y no todos son apropiados para el ejemplo de la estación climatológica. Los modelos que se discutirán en esta sección son:

1. Los *modelos de subsistemas* que muestran las agrupaciones lógicas de objetos en subsistemas coherentes. Éstos se representan utilizando una forma de los diagramas de clase en el que cada subsistema se muestra como un paquete. Los modelos de subsistemas son modelos estáticos.
2. Los *modelos de secuencia* que muestran la secuencia de interacciones de los objetos. Éstos se representan utilizando una secuencia UML o un diagrama de colaboración. Los modelos de secuencia son modelos dinámicos.
3. Los *modelos de máquinas de estado* que muestran cómo los objetos individuales cambian su estado en respuesta a los eventos. Esto se representa en UML utilizando diagramas de estado los cuales son modelos dinámicos.

Ya se han expuesto otros modelos que pueden utilizarse para diseño y análisis orientado a objetos. Los modelos de casos de uso muestran las interacciones con el sistema (Figura 14.8 y Figuras 7.6 y 7.7 del Capítulo 7); los modelos de objetos describen las clases (Figura 14.2); los modelos de generalización y herencia (Figuras 8.10, 8.11 y 8.12 del Capítulo 8) muestran cómo las clases pueden ser generalizaciones de otras clases, y los modelos de agregación (Figura 8.13) muestra cómo podemos describir colecciones de objetos.

La Figura 14.12 muestra los objetos de los subsistemas de la estación meteorológica. En esta etapa del proceso de diseño se utiliza el conocimiento del dominio de la aplicación para identificar a los objetos y servicios adicionales. En este modelo también se muestran algunas asociaciones. Por ejemplo, el objeto **CommsControler** está asociado con el objeto **WeatherStation**, y éste está asociado con el paquete **Data collection**. Esto significa que el objeto está asociado con uno o más objetos de este paquete. Un modelo de paquetes más un modelo de clases dan una descripción de las agrupaciones lógicas del sistema.

Un modelo de subsistemas es un modelo estático útil que nos muestra cómo puede estar organizado el diseño de forma lógica agrupando objetos. En la Figura 14.7 quedó reflejado este tipo de modelos, que muestra los subsistemas del sistema de mapas meteorológicos. Los paquetes UML contienen las construcciones de encapsulación y no se reflejan directamente sobre las entidades del sistema que se implementa. Sin embargo, pueden verse reflejadas en construcciones de estructuras tales como las librerías Java.

Los modelos de secuencia son modelos dinámicos que documentan, para cada modo de interacción, la secuencia de interacciones que tienen lugar entre los objetos. La Figura 14.13 es un ejemplo de modelo de secuencia que muestra las operaciones involucradas en la recogida de datos de una estación meteorológica. En un modelo de secuencia:

1. Los objetos involucrados en la interacción están ordenados horizontalmente con una línea vertical vinculada a cada objeto.

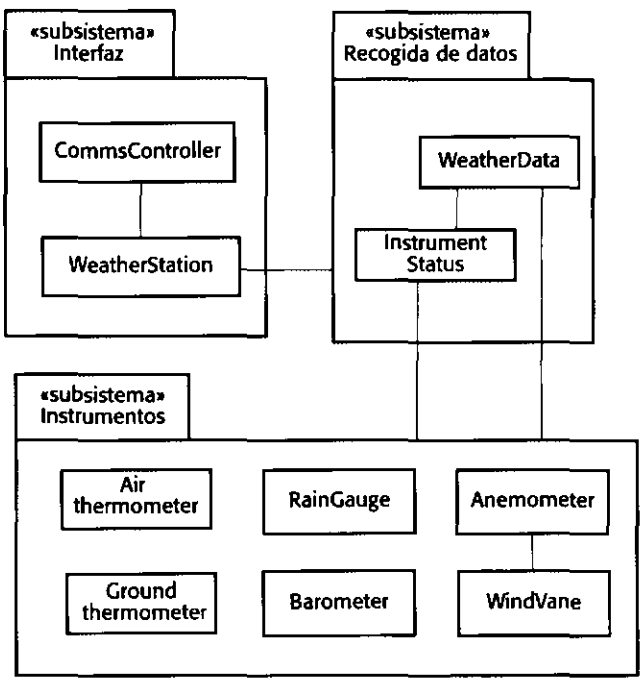


Figura 14.12 Subsistemas de una estación meteorológica.

2. El tiempo se representa verticalmente, por lo que éste avanza hacia abajo sobre las líneas punteadas. Por lo tanto, en el modelo, la secuencia de operaciones se puede leer fácilmente.
3. Las interacciones entre los objetos se representan por flechas etiquetadas que vinculan a las líneas verticales. Éstos *no* son datos que fluyen, sino que representan mensajes o eventos que son fundamentales para la interacción.
4. Los rectángulos delgados sobre la línea de vida del objeto representan el tiempo en el cual el objeto es el que tiene el control del sistema. Un objeto inicia el control en la parte superior de este rectángulo y libera el control a otro objeto en la parte inferior del mismo. Si existe una jerarquía de llamadas, el control no se libera hasta que se haya completado el último retorno a la llamada del método inicial.

Cuando se documenta un diseño, se debe producir un modelo de secuencia para cada interacción significativa. Si se ha desarrollado un modelo de casos de uso, entonces se deberá tener un modelo de secuencia para cada caso de uso identificado.

La Figura 14.13 muestra la secuencia de interacciones cuando el sistema externo de trazado de mapas solicita datos a la estación meteorológica. Este diagrama se lee como se muestra a continuación:

1. Un objeto que es una instancia de **CommsController** (**:CommsController**) recibe una petición de su entorno para enviar un informe meteorológico. Confirma que recibe la petición. La flecha con media punta indica que el emisor del mensaje no espera una contestación.
2. Este objeto envía un mensaje a un objeto que es una instancia de **WeatherStation** para crear un informe meteorológico. La instancia de **CommsControler** entonces se suspende a sí misma (su cuadro de control termina). El estilo de la cabeza de la flecha utilizada indica que la instancia del objeto **CommsController** y la instancia del objeto **WeatherStation** son objetos que se ejecutan al mismo tiempo.
3. El objeto que es una instancia de **WeatherStation** envía un mensaje al objeto **WeatherData** para resumir los datos meteorológicos. En este caso, el estilo diferente de las puntas de flecha utilizadas indica que la instancia de **WeatherStation** espera una respuesta.

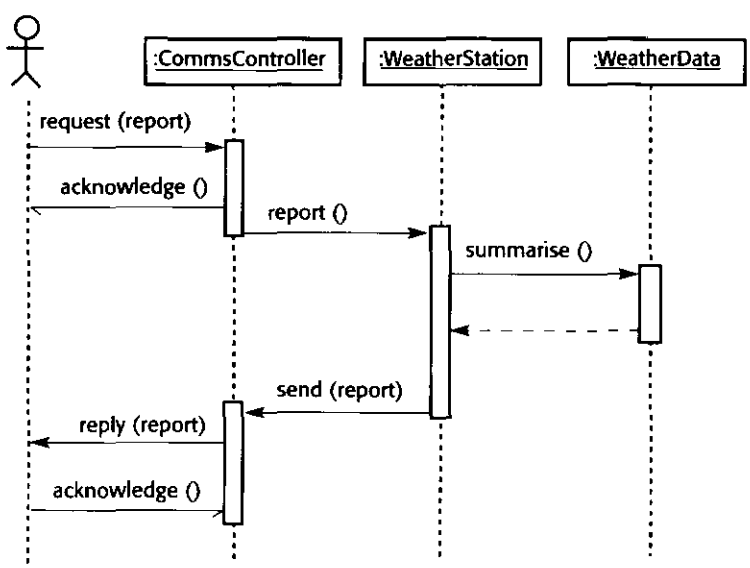


Figura 14.13 Secuencia de las operaciones de recogida de datos.

4. Se realiza el resumen y el control regresa al objeto **WeatherStation**. Las flechas punteadas indican un retorno del control.
5. Este objeto envía un mensaje a **CommsController** solicitándole transferir los datos al sistema remoto. Entonces, el objeto **WeatherStation** se suspende a sí mismo.
6. El objeto **CommsController** envía el resumen de los datos al sistema remoto, recibe una aceptación y se suspende a sí mismo esperando la siguiente solicitud.

En el diagrama de secuencia se puede ver que los objetos **CommsController** y **WeatherStation** son procesos concurrentes en los que la ejecución se puede suspender y reiniciar. En lo esencial, la instancia del objeto **CommsController** escucha los mensajes del sistema externo, decodifica estos mensajes e inicia las operaciones de la estación meteorológica.

Los diagramas de secuencia se utilizan para modelar el comportamiento combinado de un grupo de objetos, pero también son útiles para resumir el comportamiento de un solo objeto como respuesta a los diversos mensajes que puede procesar. Para hacer esto, se utiliza un modelo de máquinas de estado que muestra cómo la instancia del objeto cambia de estado dependiendo de los mensajes que reciba. UML utiliza diagramas de estado, inventados por Harel (Harel, 1987) para describir los modelos de máquinas de estado.

La Figura 14.14 es un diagrama de estado para el objeto **WeatherStation** que muestra cómo responde a la petición de varios servicios.

Este diagrama se puede leer como se muestra a continuación:

1. Si el estado del objeto es **«Shutdown»**, entonces sólo puede responder a un mensaje **startup()**. Después se mueve a un estado donde espera por mensajes adicionales. La flecha sin etiqueta con la bola negra indica que **«Shutdown»** es el estado inicial.
2. En el estado **«Waiting»**, el sistema espera por mensajes adicionales. Si se recibe un mensaje **shutdown()**, el objeto regresa al estado apagado.
3. Si se recibe un mensaje **reportWeather()**, el sistema se mueve a un estado de resumen y luego, cuando el resumen se completa, se mueve a un estado de transmisión, donde la información se transmite a través del **CommsController**. Después regresa a un estado de espera.

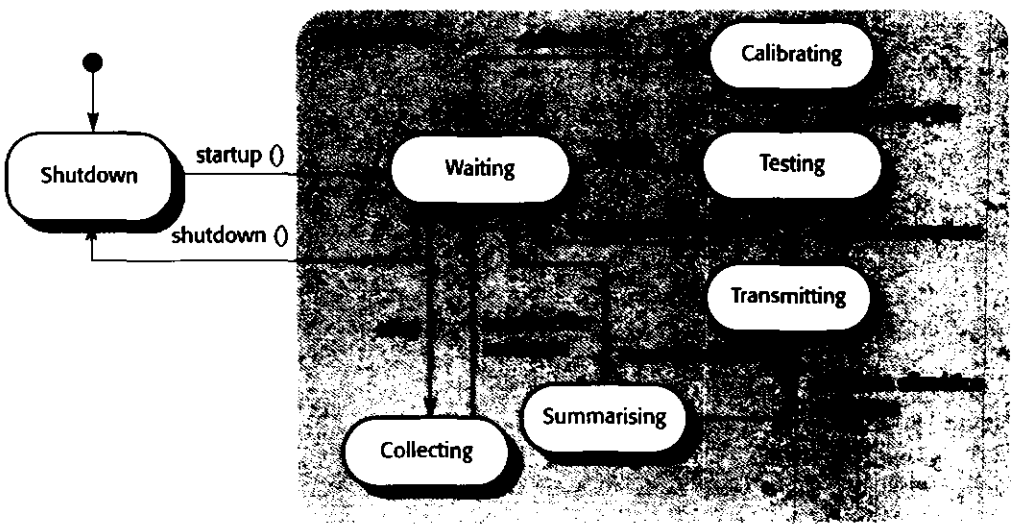


Figura 14.14 Diagrama de estado para WeatherStation.

4. Si se recibe un mensaje **calibrate()**, el sistema se mueve a un estado calibración («**Calibrate**»), después a un estado de prueba («**Testing**») y, por último, al estado de transmisión («**Transmitting**») antes de regresar al estado de espera («**Waiting**»). Si se recibe un mensaje **test()**, el sistema se mueve directamente al estado de prueba.
5. Si se recibe una señal de reloj, el sistema se mueve a un estado de recolección, donde se recogen los datos de los instrumentos. Cada instrumento está configurado para recoger sus datos.

Por lo general, no es necesario elaborar un diagrama de estado para todos los objetos que se hayan definido. Muchos de los objetos de un sistema son relativamente sencillos, y un modelo de máquina de estado no ayudaría a los implementadores a comprender estos objetos.

14.2.5 Especificación de la interfaz de los objetos

Una parte importante de cualquier proceso de diseño es la especificación de las interfaces entre los diferentes componentes del diseño. Es necesario identificar las interfaces para que los objetos y otros componentes se puedan diseñar en paralelo. Una vez que la interfaz se ha especificado, los desarrolladores de otros objetos pueden suponer que la interfaz será implementada.

Los diseñadores deben evitar la información de representación de la interfaz en el diseño de la interfaz. En su lugar, se debe ocultar la representación y se deben proveer las operaciones de los objetos para acceder y actualizar los datos. Si la representación está oculta, se puede cambiar sin afectar a los objetos que utilizan estos atributos. Esto conduce a un diseño que es inherentemente más fácil de mantener. Por ejemplo, una representación en forma de vector de una pila se puede cambiar por una representación en forma de lista sin afectar a otros objetos que usen la pila. En contraste, es frecuente mostrar los atributos en un modelo de diseño estático, ya que éste es el modo más conciso de ilustrar las características esenciales de los objetos.

Esta no es necesariamente una relación 1:1 entre los objetos y las interfaces. Los mismos objetos pueden tener varias interfaces que son puntos de vista de los métodos que proveen. Esto se permite en Java de forma directa donde las interfaces se declaran independientemente de los objetos, y los objetos «implementan» las interfaces. De igual forma, se puede acceder a un grupo de objetos a través de una sola interfaz.

El diseño de interfaces de objetos comprende la especificación del detalle de la interfaz para un objeto o un grupo de objetos. Esto significa definir las firmas y semántica de los servicios proporcionados por los objetos o por un grupo de objetos. Las interfaces se especifican en UML utilizando la misma notación que en los diagramas de clases. Sin embargo, no existe una sección de atributos, y el estereotipo en UML <interfaz> se debe incluir en la parte del nombre.

Un enfoque alternativo consiste en utilizar un lenguaje de programación para definir la interfaz. Esto se ilustra en la Figura 14.15, que muestra la especificación de la interfaz en Java para la estación meteorológica. Si las interfaces son más complejas, este enfoque es más efectivo debido a que las herramientas de verificación de sintaxis en el compilador se utilizan para descubrir errores e incongruencias en la descripción de la interfaz. La descripción en Java muestra que algunos métodos pueden tomar un número de parámetros diferente. Por lo tanto, el método apagar se puede aplicar a la estación como un todo si ésta no tiene parámetros o puede apagar un solo instrumento.

```
interface WeatherStation {

  public void WeatherStation () ;

  public void startup () ;
  public void startup (Instrument i) ;

  public void shutdown () ;
  public void shutdown (Instrument i) ;

  public void reportWeather ( ) ;

  public void test () ;
  public void test ( Instrument i ) ;

  public void calibrate ( Instrument i) ;

  public int getID () ;

} //WeatherStation
```

Figura 14.15 Descripción en Java de la interfaz de la estación meteorológica.

14.3 Evolución del diseño

Después de haber tomado una decisión para desarrollar un sistema como el sistema de recogida de datos meteorológicos, es inevitable que se hagan propuestas de cambios. Una ventaja importante de un enfoque orientado a objetos para el diseño es que simplifica el problema de hacer cambios a dicho diseño. La razón de esto es que la representación del estado del objeto no influye en el diseño. Cambiar los detalles internos de un objeto no afecta a ningún otro objeto del sistema. Más aún, debido a que los objetos están débilmente acoplados, por lo regular es sencillo introducir nuevos objetos sin efectos importantes en el resto del sistema.

Para ilustrar la robustez del enfoque orientado a objetos, supongamos que a cada estación meteorológica se le agregan las capacidades de un sistema de supervisión de la contaminación. Esto implica agregar una métrica de la calidad del aire para calcular la cantidad de diversos contaminantes en la atmósfera. Las lecturas de la contaminación se transmiten al mismo tiempo que los datos meteorológicos. Para modificar el diseño se deben hacer los siguientes cambios:

1. Debe introducirse una clase de objetos denominada **AirQuality** como parte de **WeatherStation** al mismo nivel que **WeatherData**.
2. Se debe agregar a **WeatherStation** una operación **reportAirQuality** para enviar la información de la contaminación a la computadora central. El software de control de la estación meteorológica se debe modificar para que las lecturas de la contaminación se recojan automáticamente cuando lo solicite el objeto de alto nivel **WeatherStation**.
3. Se deben agregar los objetos que representan a los tipos de instrumentos de supervisión de la contaminación. En este caso, se pueden medir los niveles de óxido nítrico, humo y benzeno.

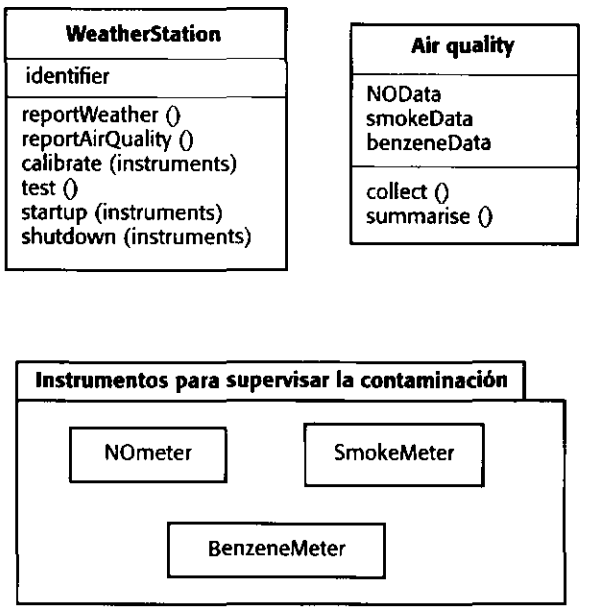


Figura 14.16
Nuevos objetos para ayudar a la supervisión de la contaminación.

Los objetos de supervisión de la contaminación están encapsulados en un paquete diferente llamado **Instrumentos para supervisar la contaminación**. Este tiene asociaciones con **AirQuality** y **WeatherSation**, pero no con los objetos de **WeatherData**. La Figura 14.16 muestra **WeatherStation** y los nuevos objetos agregados al sistema. Además de los cambios en niveles altos del sistema (**WeatherStation**) no se requiere ninguno más en el software de los objetos de la estación meteorológica. La adición de la recogida de los datos de contaminación no afecta de ninguna forma a la recogida de datos meteorológicos.

PUNTOS CLAVE

- **El diseño orientado a objetos es un medio para diseñar software de tal forma que los componentes del diseño representen objetos con su estado privado y operaciones propias en lugar de funciones.**
- **Un objeto debe tener operaciones constructor y de inspección que permitan que su estado se inspeccione y modifique. El objeto suministra servicios (operaciones que utilizan información del estado) a otros objetos. Los objetos se crean en tiempo de ejecución utilizando una especificación en una definición de clases.**
- **Los objetos se implementan de manera secuencial o concurrente. Un objeto concurrente puede ser un objeto pasivo cuyo estado sólo se cambia a través de su interfaz o un objeto activo que cambia su propio estado sin intervención externa.**
- **El Lenguaje Unificado de Modelado (UML) suministra un conjunto de notaciones diversas que se pueden utilizar para documentar un diseño orientado a objetos.**
- **El proceso de un diseño orientado a objetos incluye actividades para diseñar la arquitectura del sistema, identificar los objetos en el sistema, describir el diseño utilizando diversos modelos de objetos y documentar las interfaces de objetos.**

- **Durante un proceso de diseño orientado a objetos se produce una gran variedad de modelos, entre los que se encuentran los modelos estáticos (modelos de clases, de generalización, de asociación) y dinámicos (modelos de secuencia y de máquina de estados).**
- **Las interfaces de objetos deben definirse de forma precisa para que puedan ser utilizadas por otros objetos. Para documentar las interfaces de objetos se utiliza un lenguaje de programación como Java.**
- **Una ventaja importante del diseño orientado a objetos es que simplifica la evolución del sistema.**

LECTURAS ADICIONALES

Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd ed. Una buena introducción sobre el uso de UML dentro de un proceso de diseño orientado a objetos. Incluye los patrones de diseño tratados en el Capítulo 18. (C. Larman, 2001, Prentice Hall.)

The Unified Modeling Language User Guide. Un texto excepcional sobre UML y su utilización en la descripción de diseños orientados a objetos. Existen dos textos asociados; uno es un manual de referencia de UML, y el otro propone un proceso de desarrollo orientado a objetos. (G. Booch *et al.*, 1999, Addison-Wesley.)

A mediados de 2003 se terminó un nuevo estándar de UML (UML 2.0), pero de momento los libros no han reflejado estos cambios. Es de esperar que en 2005 estén disponibles ediciones que incorporen este estándar.

Hay un cantidad inmensa de introducciones y tutoriales de UML en la web. En la página web de libro he incluido algunos enlaces.

EJERCICIOS

14.1 Explique por qué adoptar un enfoque de diseño basado en objetos con acoplamiento débil que oculta información acerca de su representación, conduce a un diseño que puede ser fácilmente modificable.

14.2 Explique la diferencia entre un objeto y una clase, utilizando ejemplos.

14.3 ¿En qué circunstancias sería apropiado desarrollar un diseño donde los objetos se ejecuten al mismo tiempo?

14.4 Utilizando la notación gráfica de UML para clases, diseñe las siguientes clases identificando los atributos y las operaciones. Utilice la experiencia propia para decidir sobre los atributos y operaciones que están asociadas a estos objetos:

- un teléfono;
- una impresora para una computadora personal;
- un sistema estéreo personal;
- una cuenta bancaria;
- un catálogo de biblioteca.

14.5 Desarrolle el diseño detallado de la estación meteorológica proponiendo descripciones de la interfaz de los objetos mostrados en la Figura 14.11. Éste se puede expresar en Java, en C++ o en UML.

14.6 Desarrolle el diseño de la estación meteorológica para mostrar la interacción entre el subsistema de recogida de datos y los instrumentos que recogen los datos meteorológicos. Utilice diagramas de secuencia para mostrar esta interacción.

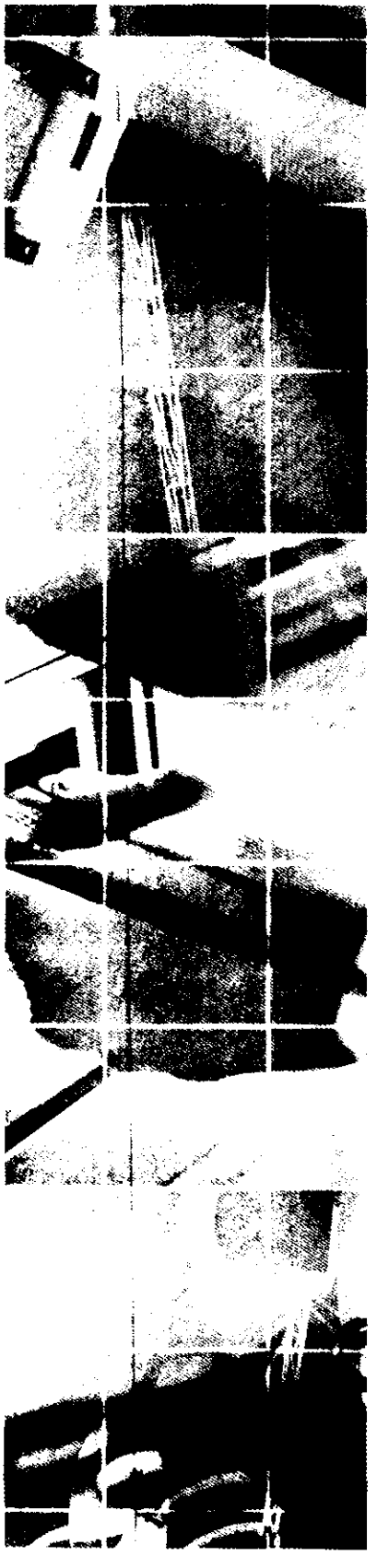
14.7 Identifique los objetos posibles en los siguientes sistemas y desarrolle un diseño orientado a objetos para ellos. Puede hacer suposiciones razonables acerca de los sistemas al derivar el diseño.

- Un sistema para administrar las agendas y el tiempo de un grupo que pretende apoyar la programación de reuniones y citas de un grupo de colaboradores. Cuando se da una cita que comprende a varias personas, el sistema encuentra un hueco común en cada una de las agendas y confirma la cita para esa hora. Si no existen huecos disponibles, interactúa con los usuarios para arreglar sus agendas personales con el fin de hacer un hueco para la cita.
- Una gasolinera va a operar completamente automatizada. Los conductores pasan sus tarjetas de crédito por un lector situado en el surtidor; la tarjeta se verifica mediante una comunicación con la computadora de la compañía de crédito, y se establece un límite de gasolina. Después el conductor reposta la cantidad de combustible deseado. Cuando se completa la entrega y se cuelga la manguera de la bomba, se le aplica el cargo del coste de la gasolina suministrada a la tarjeta de crédito del conductor. Se le devuelve la tarjeta de crédito después de aplicar el cargo. Si la tarjeta no es válida, ésta es devuelta antes de suministrar la gasolina.

14.8 Redacte las definiciones precisas de las interfaces en Java o en C++ para los objetos definidos en el ejercicio 14.7.

14.9 Dibuje un diagrama de secuencia que muestre las interacciones de los objetos en un sistema que maneja las agendas de un grupo cuando un grupo de personas tratan de concertar una cita.

14.10 Dibuje un diagrama de estado que muestre los cambios de estado posibles en uno o más objetos definidos en el Ejercicio 14.7.



15 Diseño de software de tiempo real

Objetivos

Los objetivos de este capítulo son introducir las técnicas usadas en el diseño de sistemas de tiempo real y describir algunas arquitecturas genéricas de sistemas de tiempo real. Cuando haya leído este capítulo:

- comprenderá el concepto de sistema de tiempo real y por qué los sistemas de tiempo real se implementan normalmente como un conjunto de procesos concurrentes;
- habrá sido introducido en el proceso de diseño de sistemas de tiempo real;
- comprenderá el papel que desempeña un sistema operativo de tiempo real;
- conocerá las arquitecturas de procesos genéricos para sistemas de monitorización y control y sistemas de adquisición de datos.

Contenidos

Las computadoras se utilizan para controlar una amplia variedad de sistemas que van desde máquinas domésticas sencillas hasta plantas enteras de fabricación. Estas computadoras interactúan directamente con dispositivos hardware. El software de dichos sistemas es *software de tiempo real embebido* que debe reaccionar a eventos generados por el hardware y emitir señales de control como respuesta a estos eventos. Está embebido en sistemas hardware más grandes y debe responder, en tiempo real, a eventos del entorno del sistema.

Los sistemas de tiempo real embebidos son diferentes de otros tipos de sistemas software. Su correcto funcionamiento depende de que el sistema responda a los eventos dentro de un corto intervalo de tiempo. Se puede definir un sistema de tiempo real como sigue:

> *Un sistema de tiempo real es un sistema software cuyo correcto funcionamiento depende de los resultados producidos por el mismo y del instante de tiempo en el que se producen estos resultados. Un sistema de tiempo real blando (soft) es un sistema cuyo funcionamiento se degrada si los resultados no se producen de acuerdo con los requerimientos temporales especificados. Un sistema de tiempo real duro (hard) es un sistema cuyo funcionamiento es incorrecto si los resultados no se producen de acuerdo con la especificación temporal.*

Una respuesta a tiempo es un factor importante en todos los sistemas embebidos, pero, en algunos casos, no es necesaria una respuesta muy rápida. Por ejemplo, el sistema de bomba de insulina que se utiliza como ejemplo en varios capítulos de este libro es un sistema embebido. Sin embargo, aunque se necesita comprobar el nivel de glucosa a intervalos periódicos, no es necesario responder muy rápidamente a los eventos externos. Por lo tanto, se utilizan ejemplos diferentes en este capítulo para ilustrar los fundamentos de diseño de los sistemas de tiempo real.

Una forma de ver un sistema de tiempo real es como un sistema de estímulo/respuesta. Dado un determinado estímulo de entrada, el sistema debe producir la correspondiente salida. Se puede, por lo tanto, definir el comportamiento de un sistema de tiempo real haciendo una lista de los estímulos recibidos por el sistema, las respuestas asociadas y el tiempo en el que dichas respuestas deben producirse.

Los estímulos pueden pertenecer a dos clases:

1. *Estímulos periódicos*. Ocurren a intervalos de tiempo predecibles. Por ejemplo, el sistema debe examinar un sensor cada 50 milisegundos y realizar una acción (respuesta) dependiendo del valor de ese sensor (estímulo).
2. *Estímulos aperiódicos*. Ocurren de forma irregular. Normalmente son provocados utilizando el mecanismo de interrupciones de la computadora. Un ejemplo de dicho estímulo podría ser una interrupción para indicar que una transferencia de E/S se ha completado y que los datos están disponibles en un búfer.

Los estímulos periódicos en un sistema de tiempo real son generados normalmente por sensores asociados al sistema. Éstos proporcionan información sobre el estado del entorno del sistema. Las respuestas son dirigidas a un conjunto de actuadores que controlan algún equipo, como una bomba, que influye en el entorno del sistema. Los estímulos aperiódicos pueden generarse por actuadores o por sensores. A menudo indican alguna condición excepcional, como un fallo en el hardware, que debe ser manejado por el sistema. Este modelo sensor-sistema-actuador de un sistema de tiempo real embebido se ilustra en la Figura 15.1.

Un sistema de tiempo real tiene que responder a estímulos que ocurren en diferentes instantes de tiempo. Por lo tanto, se tiene que organizar su arquitectura para que, tan pronto como se reciba un estímulo, el control sea transferido al manejador adecuado. Esto no es práctico

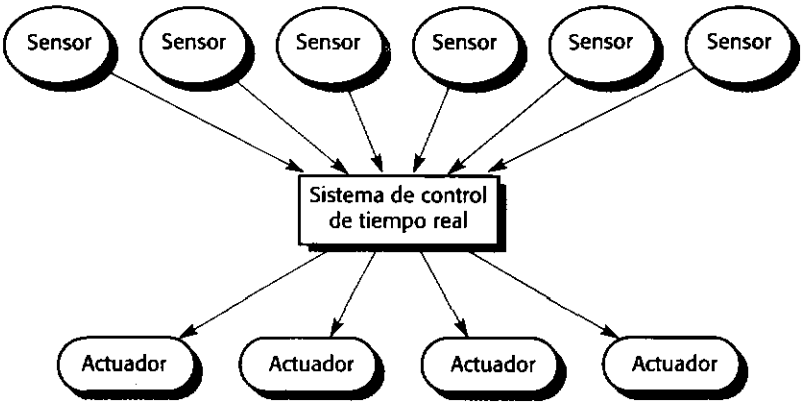


Figura 15.1 Modelo general de un sistema de tiempo real.

en programas secuenciales. Por consiguiente, los sistemas de tiempo real se diseñan como un conjunto de procesos concurrentes que cooperan entre sí. Con el objeto de soportar la gestión de estos procesos, la plataforma de ejecución para la mayoría de los sistemas de tiempo real incluye un *sistema operativo de tiempo real*. Las facilidades que proporciona este sistema operativo son accedidas a través del sistema de soporte en tiempo de ejecución (*run-time system*) para el lenguaje de programación de tiempo real utilizado.

La generalidad de este modelo estímulo-respuesta de un sistema de tiempo real conduce a un modelo arquitectónico genérico abstracto en el que hay tres tipos de procesos. Para cada tipo de sensor, hay un proceso de gestión del sensor; los procesos computacionales calculan la respuesta requerida para el estímulo recibido por el sistema; los procesos de control de actuadores controlan el funcionamiento del actuador. Este modelo permite recoger rápidamente los datos desde el sensor (antes de que la siguiente entrada esté disponible) y permite que su procesamiento y la respuesta asociada al actuador se realicen más tarde.

La arquitectura genérica puede instanciarse a varias arquitecturas de aplicaciones diferentes que amplían el conjunto de arquitecturas estudiadas en el Capítulo 13. Las arquitecturas de aplicaciones de tiempo real son instancias de la arquitectura conducida por eventos en la cual el estímulo, directa o indirectamente, provoca la generación de eventos. En este capítulo, se introducen dos arquitecturas de aplicaciones más: la arquitectura para sistemas de monitorización y control (en la Sección 15.3), y la arquitectura para sistemas de adquisición de datos.

Los lenguajes de programación desarrollados para sistemas de tiempo real tienen que incluir facilidades para acceder al hardware del sistema, y debería ser posible predecir la duración de operaciones particulares realizadas en estos lenguajes. Los sistemas de tiempo real duros todavía se programan algunas veces en ensamblador para que puedan cumplirse los estrechos plazos de tiempo (*deadline*). Los lenguajes a nivel de sistemas, como C, que permiten generar código eficiente también se utilizan en general.

La ventaja de utilizar un lenguaje de programación de sistemas de bajo nivel como C es que permite el desarrollo de programas muy eficientes. Sin embargo, estos lenguajes no incluyen construcciones para soportar la concurrencia o la gestión de recursos compartidos. Éstas se implementan a través de llamadas al sistema operativo de tiempo real que no pueden ser comprobadas por el compilador, de forma que los errores de programación son más probables. Los programas son también a menudo más difíciles de comprender debido a que las características de tiempo real no están explícitas en el programa.

Desde hace pocos años, se ha venido realizando un amplio trabajo a fin de extender Java para el desarrollo de sistemas de tiempo real (Nilsen, 1998; Higuera-Toledano e Issarny, 2000; Hardin *et al.*, 2002). Este trabajo implica la modificación del lenguaje para tratar los principales problemas de tiempo real:

1. No es posible especificar el instante de tiempo en el que los hilos se deberían ejecutar.
2. La recolección de basura es incontrolable: puede empezar en cualquier momento. Por lo tanto, el comportamiento temporal de los hilos es impredecible.
3. No es posible descubrir los tamaños de las colas asociadas con recursos compartidos.
4. La implementación de la Máquina Virtual de Java varía de una computadora a otra, de forma que el mismo programa puede tener diferentes comportamientos temporales.
5. El lenguaje no permite un análisis detallado de la memoria o del procesador en tiempo de ejecución.
6. No hay una forma estándar de acceder al hardware del sistema.

Las versiones de tiempo real de Java, como J2ME de Sun (Java 2 Micro Edition), están actualmente disponibles. Varios vendedores suministran implementaciones de la Máquina Virtual de Java adaptada al desarrollo de sistemas de tiempo real. Estos desarrollos implican que Java se usará cada vez más como lenguaje de programación en tiempo real.

15.1 Diseño del sistema

Tal y como se indicó en el Capítulo 2, parte del proceso de diseño implica decidir qué capacidades del sistema tienen que implementarse en software y cuáles en hardware. Para muchos sistemas de tiempo real embebidos en productos de consumo, como los sistemas en aparatos telefónicos, los costes y la potencia de consumo del hardware, son críticos. Se pueden utilizar procesadores específicos diseñados para soportar sistemas embebidos y, para algunos sistemas, tiene que diseñarse y construirse hardware de propósito específico.

Esto significa que un proceso de diseño desde arriba hacia abajo —en el que el diseño comienza con un modelo abstracto que se va descomponiendo y desarrollando en una serie de etapas— no es práctico para la mayoría de los sistemas de tiempo real. Las decisiones de bajo nivel sobre el hardware, el software de soporte y las cuestiones temporales sobre el sistema deben considerarse en etapas tempranas del proceso de desarrollo. Esto limita la flexibilidad de los diseñadores del sistema y puede implicar que se requieran funcionalidades software adicionales, como la gestión de la batería y del suministro eléctrico.

Los eventos (estímulos) deberían ser elementos centrales del proceso de diseño de software de tiempo real en lugar de los objetos o funciones. Hay varias etapas intercaladas en este proceso de diseño:

1. Identificar los estímulos que el sistema debe procesar y las respuestas asociadas.
2. Para cada estímulo y respuesta asociada, identificar las restricciones temporales que se aplican tanto al procesamiento del estímulo como al de la respuesta.
3. Elegir una plataforma de ejecución para el sistema: el hardware y el sistema operativo de tiempo real que se va a utilizar. Entre los factores que influyen en esta elección se encuentran las restricciones temporales del sistema, las limitaciones de energía disponible, la experiencia del grupo de desarrollo y el coste de la máquina en la que se va a ejecutar el sistema entregado.

4. Incorporar el procesamiento de estímulos y respuestas a varios procesos concurrentes. Una buena regla en el diseño de sistemas de tiempo real es asociar un proceso con cada tipo de estímulo y respuesta tal y como se muestra en la Figura 15.2.
5. Para cada estímulo y respuesta, diseñar algoritmos para llevar a cabo los cálculos requeridos. Los diseños de los algoritmos tienen que desarrollarse a menudo relativamente pronto en el proceso de diseño para proporcionar una indicación de la cantidad de procesamiento requerido y del tiempo necesario para completar dicho procesamiento.
6. Diseñar un sistema de planificación de los procesos que asegure que dichos procesos comienzan a tiempo para cumplir sus plazos de ejecución.

El orden de estas actividades en el proceso depende del tipo de sistema que se esté desarrollando y de los requerimientos de su proceso y plataforma. En algunos casos, se puede seguir una aproximación completamente abstracta en la que se empieza con los estímulos y el procesamiento asociado y se decide más tarde en el proceso el hardware y las plataformas de ejecución. En otros casos, la elección del hardware y el sistema operativo se realiza antes de que comience el diseño del software, y se tiene que orientar el diseño según las capacidades del hardware.

Los procesos en un sistema de tiempo real tienen que coordinarse. Los mecanismos de coordinación de procesos aseguran la exclusión mutua de recursos compartidos. Cuando un proceso modifica un recurso compartido, al resto de los procesos no se les debería permitir cambiar ese recurso. Los mecanismos para asegurar la exclusión mutua comprenden semáforos (Dijkstra, 1968), monitores (Hoare, 1974) y regiones críticas (Brinch-Hansen, 1973). Estos mecanismos se describen en la mayoría de los textos sobre sistemas operativos (Tanenbaum, 2001; Silberschatz, *et al.*, 2002).

Una vez que se ha elegido la plataforma de ejecución para el sistema, se ha diseñado una arquitectura para el proceso y se ha decidido una política de planificación (*scheduling*), puede necesitarse comprobar que el sistema satisface sus requerimientos temporales. Se puede hacer esto mediante el análisis estático del sistema utilizando conocimientos sobre el comportamiento temporal de los componentes, o también mediante simulación. Este análisis puede revelar que el sistema no funcionará de forma adecuada. Entonces, la arquitectura del proceso, la política de planificación del procesador, la plataforma de ejecución o todos ellos pueden tener que ser rediseñados para mejorar el rendimiento del sistema.

El análisis temporal para sistemas de tiempo real es difícil. Debido a que los estímulos aperiódicos son impredecibles, los diseñadores tienen que hacer suposiciones sobre la probabilidad de ocurrencia de estos estímulos (y por lo tanto del servicio requerido) en un instante de tiempo concreto. Estas suposiciones pueden ser incorrectas, y el rendimiento del sistema después de la entrega puede no ser el adecuado. El libro de Cooling (Cooling, 2003) describe las técnicas para el análisis de rendimiento de sistemas de tiempo real.

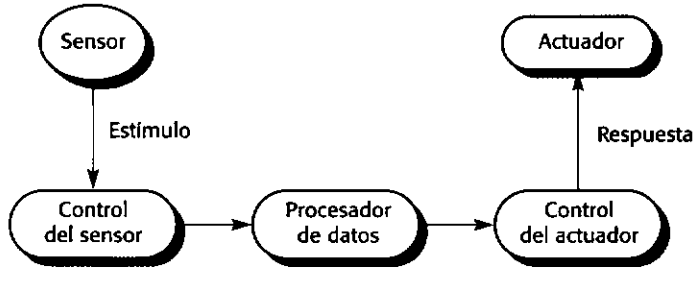


Figura 15.2
Procesos de control sensor/actuador.

Debido a que los sistemas de tiempo real deben satisfacer sus restricciones temporales, es posible que no se pueda usar el desarrollo orientado a objetos para sistemas de tiempo real duros. El desarrollo orientado a objetos implica la ocultación de representaciones de datos y el acceso a los valores de los atributos a través de operaciones definidas en el objeto. Esto significa que hay una sobrecarga significativa que afecta al rendimiento en sistemas orientados a objetos, debido a que se necesita código extra para acceder a los atributos y para manejar las llamadas a las operaciones. La consecuente pérdida de rendimiento puede hacer imposible el satisfacer los plazos temporales de tiempo real.

Las restricciones temporales u otros requerimientos pueden implicar a veces que es mejor implementar algunas funciones del sistema, tales como el procesamiento de la señal, en hardware en vez de en software. Los componentes hardware consiguen un rendimiento mucho mejor que su equivalente software. Pueden identificarse los cuellos de botella en el procesamiento de sistemas y ser sustituidos por hardware, con lo que se evitan optimizaciones del software caras.

15.1.1 Modelado de sistemas de tiempo real

Los sistemas de tiempo real deben responder a eventos que tienen lugar a intervalos irregulares. Estos eventos (o estímulos) a menudo hacen que el sistema cambie a un estado diferente. Por esta razón, el modelado de máquina de estados, descrito en el Capítulo 8, se utiliza a menudo para modelar sistemas de tiempo real.

Los modelos de máquina de estados son una buena aproximación independiente del lenguaje de representar el diseño de un sistema de tiempo real y, por lo tanto, son una parte integral de los métodos de diseño de sistemas de tiempo real (Gomaa, 1993). UML soporta el desarrollo de modelos de estados basados en diagramas de estado (Harel, 1987; Harel, 1988). Los diagramas de estado estructuran los modelos de estados para que los grupos de estados puedan considerarse como una única entidad. Douglass analiza el uso de UML en el desarrollo de sistemas de tiempo real (Douglass, 1999).

Un modelo de estados de un sistema supone que, en cualquier momento, el sistema está en uno de varios estados posibles. Cuando se recibe un estímulo, éste puede producir una transición a un estado diferente. Por ejemplo, un sistema que controla una válvula puede pasar desde un estado «Válvula abierta» a un estado «Válvula cerrada» cuando se recibe una orden (el estímulo) del operador.

Ya se ha ilustrado esta aproximación para el modelado de sistemas en el Capítulo 8 utilizando el modelo de un sencillo horno microondas. La Figura 15.3 es otro ejemplo de un modelo de máquina de estados que muestra el funcionamiento de un sistema software de suministro de combustible en una bomba de petróleo (gas). Los rectángulos redondeados representan estados del sistema, y las flechas representan estímulos que fuerzan una transición de un estado a otro. Los nombres elegidos en el diagrama de máquina de estados son descriptivos y la información asociada indica las acciones realizadas por los actuadores del sistema o la información que es visualizada.

El sistema de suministro de combustible se diseña para permitir un funcionamiento automático. El comprador inserta una tarjeta de crédito en un lector de tarjetas incluido en la bomba. Esto provoca una transición a un estado **Leyendo** en donde los detalles de la tarjeta son leídos y se le solicita al comprador que retire la tarjeta. El sistema se mueve al estado **Validando** en donde la tarjeta es validada. Si la tarjeta es válida, el sistema inicializa la bomba y, cuando la manguera del combustible es retirada de su soporte, está lista para suministrar el combusti-

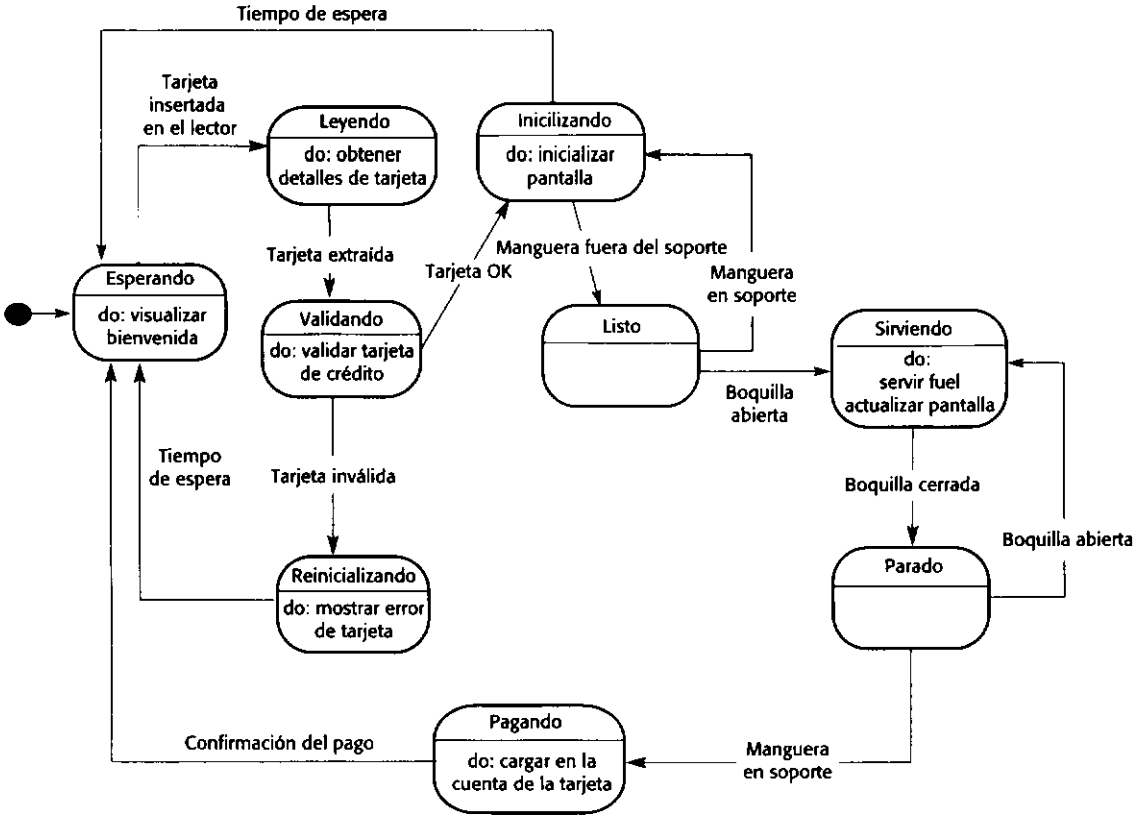


Figura 15.3 Modelo de máquina de estados de una bomba de petróleo (gas).

ble. Activando el gatillo de la boquilla de la manguera se bombea el combustible; éste se detiene cuando se suelta el gatillo (por simplificar, no se ha tenido en cuenta el interruptor de presión diseñado para detener un posible derrame del combustible). Cuando se ha completado el suministro de combustible y el comprador ha vuelto a colocar la manguera en su soporte, el sistema pasa al estado **Pagando** en donde se realiza un cargo en la cuenta del cliente.

15.2 Sistemas operativos de tiempo real

Todos los sistemas de tiempo real, incluso hasta los sistemas embebidos más sencillos, trabajan hoy en día conjuntamente con un sistema operativo de tiempo real (RTOS). Un sistema operativo de tiempo real gestiona los procesos y asignación de recursos en un sistema de tiempo real. Lanza y detiene los procesos para que los estímulos puedan ser manejados y asigna memoria y recursos del procesador. Hay muchos productos RTOS disponibles, desde sistemas sencillos muy pequeños para dispositivos de consumo, hasta sistemas complejos para teléfonos y dispositivos móviles, y sistemas operativos específicamente diseñados para control de procesos y telecomunicaciones.

Los componentes de un RTOS (Figura 15.4) dependen del tamaño y complejidad del sistema de tiempo real que se esté desarrollando. Normalmente, exceptuando los sistemas más sencillos, todos incluyen:

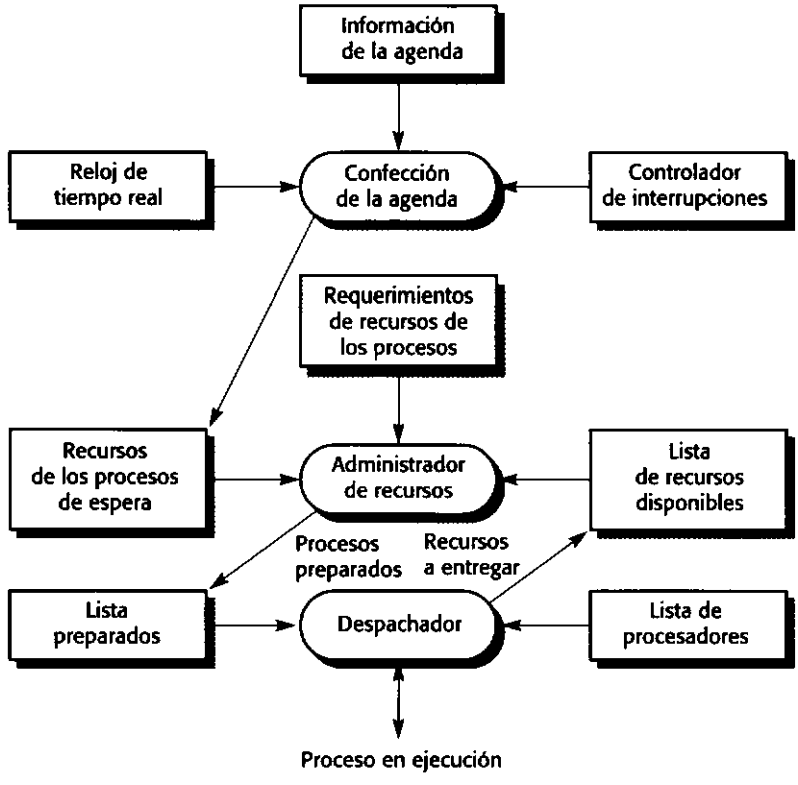


Figura 15.4
Componentes de un sistema operativo de tiempo real.

1. *Un reloj de tiempo real.* Proporciona información para planificar los procesos de forma periódica.
2. *Un manejador de interrupciones.* Gestiona las solicitudes aperiódicas de los servicios.
3. *Un planificador.* Este componente se encarga de examinar los procesos que pueden ser ejecutados y elegir uno de ellos para su ejecución.
4. *Un gestor de recursos.* Dado un proceso que es planificado para ejecución, el gestor de recursos asigna la memoria adecuada y los recursos del procesador.
5. *Un despachador.* Este componente tiene la función de iniciar la ejecución de un proceso.

Los sistemas operativos de tiempo real para sistemas grandes, tales como control de procesos o sistemas de telecomunicaciones, pueden tener facilidades adicionales, tales como gestión de almacenamiento en disco y gestión de fallos, que detectan e informan de fallos del sistema y un gestor de configuraciones que soporta la reconfiguración dinámica de aplicaciones de tiempo real.

15.2.1 Gestión de procesos

Los sistemas de tiempo real tienen que manejar eventos externos rápidamente y, en algunos casos, satisfacer plazos de tiempo para el procesamiento de estos eventos. Esto significa que los procesos de manejo de eventos deben ser planificados a tiempo para su ejecución y así detectar el evento, y se deben asignar suficientes recursos del procesador para satisfacer su pla-

zo de tiempo. El gestor de procesos en un RTOS es responsable de elegir los procesos para su ejecución, asignar el procesador y recursos de memoria, e iniciar y detener la ejecución de un proceso sobre un procesador.

El gestor de procesos tiene que gestionar procesos con diferentes prioridades. Para algunos estímulos, como los asociados con ciertos eventos excepcionales, es esencial que su procesamiento sea completado dentro de los límites de tiempo especificados. Otros procesos pueden retrasarse de forma segura si un proceso más crítico requiere el servicio. Como consecuencia, los RTOS tienen que ser capaces de gestionar al menos dos niveles de prioridad para los procesos del sistema:

1. *Nivel de interrupción.* Es el nivel de prioridad más alto. Se asigna a procesos que necesitan una respuesta muy rápida. Uno de estos procesos será el proceso del reloj de tiempo real.
2. *Nivel de reloj.* Este nivel de prioridad se asigna a los procesos periódicos.

Puede haber un nivel más de prioridad asignado a los procesos que se ejecutan en un segundo plano (tales como un proceso de autocomprobación) que no tienen un plazo límite de terminación. Estos procesos son planificados para su ejecución cuando el procesador esté ocioso.

Dentro de cada uno de estos niveles de prioridad, se pueden asignar diferentes prioridades a distintos tipos de procesos. Por ejemplo, pueden existir varias líneas de interrupción. Una interrupción de un dispositivo muy rápido puede interrumpir el procesamiento de una interrupción de un dispositivo más lento para evitar la pérdida de información. La asignación de prioridades de procesos para que todos ellos sean atendidos a tiempo requiere normalmente un extenso análisis y simulación.

Los procesos periódicos son procesos que deben ejecutarse a intervalos de tiempo predefinidos para la adquisición de datos y el control de los actuadores. En la mayoría de los sistemas de tiempo real, habrá varios tipos de procesos periódicos. Éstos tendrán diferentes periodos (el tiempo transcurrido entre ejecuciones del proceso), tiempos de ejecución y plazos de tiempo (el tiempo en el cual se debe completar el procesamiento). Utilizando los requerimientos temporales especificados en el programa de la aplicación, el RTOS ordena la ejecución de los procesos periódicos para que todos ellos puedan cumplir sus plazos de tiempo.

Las acciones llevadas a cabo por el sistema operativo para la gestión de procesos periódicos se muestran en la Figura 15.5. El planificador examina la lista de procesos periódicos y selecciona un proceso para su ejecución. La elección depende de la prioridad de los procesos, de los periodos de los procesos, de los tiempos de ejecución esperados y de los plazos de tiempo de los procesos listos para ejecución. Algunas veces, dos procesos con diferentes plazos de tiempo deberían ejecutarse en el mismo tic del reloj. En tal situación, un proceso debe retrasarse de forma que su plazo de tiempo todavía pueda cumplirse.

Los procesos que tienen que responder a eventos asíncronos son normalmente conducidos por interrupciones. El mecanismo de interrupciones de la computadora hace que el control se transfiera a una posición de memoria predeterminada. Esta posición contiene una instrucción para saltar a una rutina de servicio de interrupciones sencilla y más rápida. La rutina de servicio de interrupciones primero inhabilita las interrupciones para evitar ser interrumpida.

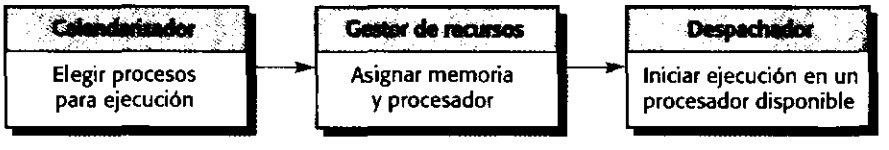


Figura 15.5
Acciones del RTOS requeridas para iniciar un proceso.

Seguidamente, encuentra la causa de la interrupción e inicia, con la prioridad más alta, un proceso que maneja los estímulos que provocan la interrupción. En algunos sistemas de adquisición de datos de alta velocidad, el manejador de interrupciones guarda los datos marcados por la interrupción para su procesamiento posterior. A continuación, las interrupciones se habilitan de nuevo y el control se devuelve al sistema operativo.

En cualquier momento, pueden existir varios procesos, con diferentes prioridades, que podrían ejecutarse. El planificador del proceso implementa las políticas de planificación del sistema que determinan el orden de la ejecución de los procesos. Hay dos estrategias fundamentales de planificación:

1. *Planificador sin reemplazo.* Una vez que un proceso ha sido planificado para su ejecución, se ejecuta hasta el final o hasta que se bloquee por alguna razón, tal como la espera de una entrada. Esto puede causar problemas cuando existen procesos con diferentes prioridades y un proceso con una prioridad más alta tiene que esperar a que un proceso de menor prioridad termine.
2. *Planificación con reemplazo.* La ejecución de un proceso se puede detener si un proceso de prioridad más alta requiere el uso del procesador. El proceso de prioridad más alta reemplaza la ejecución del proceso de prioridad más baja y se le asigna el procesador.

Dentro de estas estrategias, se han desarrollado diferentes algoritmos de planificación. Éstos comprenden la planificación denominada *round-robin*, en donde cada proceso se ejecuta por turnos, la planificación de frecuencia monótona (*rate monotonic*), en donde se le da prioridad al proceso con el periodo más corto, y la estrategia de planificación consistente en ejecutar primero el proceso con el plazo de tiempo más corto (Burns y Wellings, 2001).

La información sobre los procesos a ejecutar se envía al gestor de recursos. Éste asigna memoria y, en un sistema multiprocesador, un procesador a este proceso. Después el proceso se sitúa en la lista de procesos preparados, que es una lista de procesos listos para su ejecución. Cuando un procesador termina de ejecutar un proceso y vuelve a estar disponible, se invoca al despachador. Éste examina la lista de preparados para encontrar un proceso que pueda ejecutarse en el procesador disponible e inicia su ejecución.

15.3 Sistemas de monitorización y control

Los sistemas de monitorización y control son una clase importante de sistemas de tiempo real. Éstos comprueban los sensores que proporcionan información sobre el entorno del sistema y llevan a cabo acciones dependiendo de la lectura del sensor. Los sistemas de monitorización realizan una acción cuando se detecta algún valor excepcional del sensor. Los sistemas de control controlan continuamente los actuadores hardware dependiendo del valor de los sensores asociados.

Las características de los sistemas de monitorización y control se muestran en la Figura 15.6. Cada tipo de sensor que se está monitorizando tiene su propio proceso de monitorización, así como cada tipo de actuador que se está controlando. Un proceso de monitorización recopila e integra los datos antes de enviarlos a un proceso de control, el cual toma decisiones basadas en estos datos y envía los comandos de control adecuados a los procesos de control del equipo. En sistemas simples, las responsabilidades de monitorización y control pueden integrarse en un único proceso. Aquí también se han mostrado otros dos procesos que

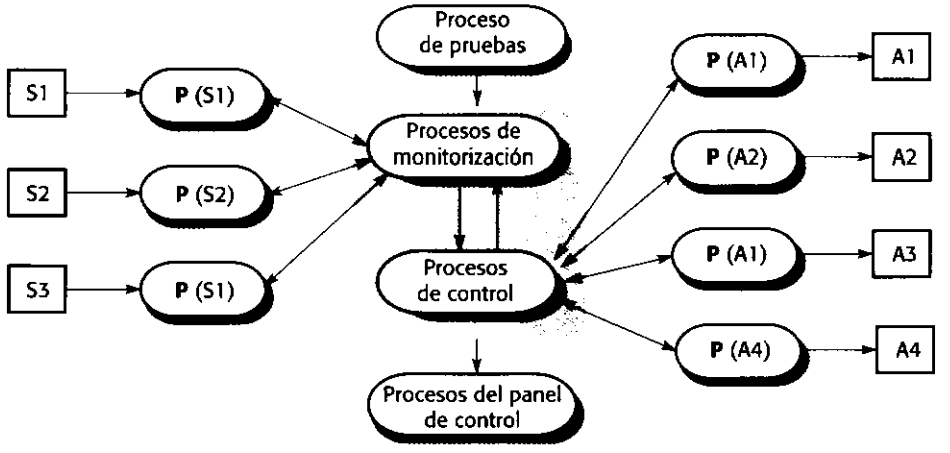


Figura 15.6 Arquitectura genérica para un sistema de monitorización y control.

pueden incluirse en sistemas de monitorización y control. Éstos son: un proceso de pruebas que puede ejecutar programas de test del hardware y un proceso de panel de control que gestiona los paneles de control del sistema o la consola del operador.

Para ilustrar el diseño de los sistemas de monitorización y control, se utiliza un ejemplo de un sistema de alarma antirrobo que podría instalarse en un edificio de oficinas:

> Se tiene que implementar un sistema software para controlar un sistema de alarma antirrobo para su instalación en edificios comerciales. Éste utiliza varios tipos de sensores, que comprenden detectores de movimiento en estancias individuales, sensores en las ventanas de la planta baja que detectan cuándo se ha abierto o roto la ventana, y sensores en las puertas de los pasillos que detectan la apertura de éstas. El sistema se compone de 50 sensores en las ventanas, 30 sensores en las puertas y 200 detectores de movimiento.
>
> Cuando un sensor detecta la presencia de un intruso, el sistema automáticamente realiza una llamada a la policía local y, utilizando un sintetizador de voz, informa de la localización de la alarma. Enciende las luces en las estancias alrededor del sensor activo y activa una alarma sonora. El sistema de sensores normalmente utiliza el suministro eléctrico general, pero además está equipado con una batería de apoyo. La pérdida de energía se detecta utilizando un monitor de circuito de energía independiente que monitoriza los principales voltajes. Éste interrumpe el sistema de alarma cuando se detecta una caída de voltaje.

Este sistema es un sistema de tiempo real «blando» que no tiene requerimientos temporales estrictos. Los sensores no necesitan detectar eventos a altas velocidades; solamente necesitan ser consultados dos veces por segundo. Para hacer que el ejemplo sea más fácil de entender, se ha simplificado el diseño dejando de lado los procesos de prueba y visualización.

El proceso de diseño sigue los pasos indicados en la Sección 15.1, por lo que se comienza identificando los estímulos aperiódicos que recibe el sistema y sus respuestas asociadas. Debido a las simplificaciones en el diseño propuesto, pueden pasarse por alto los estímulos generados por los procedimientos de prueba del sistema y las señales externas para desactivarlo en caso de un evento de falsa alarma. Esto significa que solamente hay que procesar dos tipos de estímulos:

1. *Fallo en el suministro eléctrico.* Éste se genera por el monitor del circuito. La respuesta requerida es activar el circuito de la batería de apoyo enviando señales a un interruptor electrónico de la batería.

2. *Alarma contra intrusos.* Éste es un estímulo generado por uno de los sensores del sistema. La respuesta a este estímulo es calcular el número de la estancia del sensor activo, realizar una llamada a la policía, iniciar el sintetizador de voz para efectuar la llamada, y activar la alarma sonora para intrusos y las luces del edificio en el área.

El siguiente paso en el proceso de diseño es considerar las restricciones temporales asociadas a cada estímulo y su correspondiente respuesta. Estas restricciones temporales se muestran en la Figura 15.7. Normalmente, se deberían listar las restricciones temporales para cada tipo de sensor de forma independiente, incluso cuando, como ocurre en este caso, todas sean las mismas. Gestionándolas de forma independiente, facilita futuros cambios y resulta más sencillo calcular el número de veces que el proceso de control tiene que ejecutarse cada segundo.

La asignación de las funciones del sistema a procesos concurrentes es la siguiente etapa de diseño. Hay tres tipos de sensores que deben consultarse de forma periódica, cada uno con un proceso asociado. Existe un sistema conducido por interrupciones para manejar los fallos en el suministro eléctrico y el cambio a la batería de apoyo, un sistema de comunicaciones, un sintetizador de voz, un sistema de alarma sonora y un sistema de encendido de luces para encender las luces alrededor del sensor. Un proceso independiente controla cada uno de estos sistemas. Esto conduce a la arquitectura del sistema mostrada en la Figura 15.8.

En la Figura 15.8, las flechas etiquetadas unen procesos, indicando los flujos de datos entre ellos, mientras que la etiqueta indica el tipo de flujo de datos. No todos los procesos reciben datos de otros procesos. Por ejemplo, el proceso responsable de gestionar un fallo en el suministro eléctrico no necesita de ningún proceso del sistema.

La línea asociada con cada proceso sobre su extremo superior izquierdo se utiliza para indicar cómo se controla el proceso. Las líneas sobre un proceso periódico son líneas continuas cuya etiqueta indica el número mínimo de veces que un proceso debería ejecutarse por se-

Estímulo/Respuesta	Requerimientos temporales
Interruptor de fallos en el suministro eléctrico	El cambio al suministro eléctrico de apoyo debe completarse en un plazo máximo de 50 ms.
Alarma de puerta	Cada alarma de puerta debería ser consultada dos veces por segundo.
Alarma de ventana	Cada alarma de ventana debería ser consultada dos veces por segundo.
Detector de movimiento	Cada detector de movimiento debería ser consultado dos veces por segundo.
Alarma sonora	La alarma sonora debería ser activada en medio segundo después de que una alarma sea activada por un sensor
Interruptor de luces	Las luces deberían ser encendidas en medio segundo a partir de que una alarma sea activada por un sensor.
Comunicaciones	La llamada a la policía debería comenzar a los dos segundos de que una alarma sea activada por un sensor.
Sintetizador de voz	Un mensaje sintetizado debería estar disponible a los cuatro segundos de que una alarma sea activada por un sensor.

Figura 15.7 Requerimientos temporales del estímulo/respuesta.

gundo. Los procesos aperiódicos tienen líneas discontinuas sobre su extremo superior izquierdo, etiquetadas con el evento que hace que el proceso se planifique.

El número de sensores que tienen que ser consultados y los requerimientos temporales del sistema se utilizan para calcular la periodicidad con la que cada proceso tiene que ser planificado. Por ejemplo, hay 30 sensores de puertas que deben ser comprobados dos veces por segundo. Esto significa que el proceso asociado al sensor de la puerta debe ejecutarse 60 veces por segundo (60 Hz). El proceso del detector de movimiento debe ejecutarse 400 veces por segundo debido a que hay 200 sensores de movimiento en el sistema. La información de control sobre los procesos del actuador (por ejemplo, el controlador de la alarma sonora, el controlador de las luces, etc.) indica que han sido activados por una orden explícita del proceso **Sistema de alarma** o por una **Interrupción de fallo de suministro eléctrico**.

Éstos pueden implementarse en Java utilizando hilos de ejecución. La Figura 15.9 muestra el código Java que implementa el proceso **BuildingMonitor**, que realiza una consulta a los sensores del sistema. Si éstos detectan un intruso, el software activa el sistema de alarma asociado. Aquí se utiliza Java estándar y se supone que los requerimientos temporales (incluidos como comentarios) pueden cumplirse. Tal y como se indicó anteriormente, el lenguaje Java en su forma estándar no incluye facilidades para permitir la especificación de la frecuencia de ejecución de los hilos.

Una vez que ha sido definida la arquitectura de los procesos del sistema, deberían diseñarse los algoritmos para el procesamiento de los estímulos y la generación de respuestas. Tal y como se explicó en la Sección 15.1, esta etapa de diseño detallado es necesaria al principio del proceso de diseño para asegurar que el sistema pueda cumplir sus restricciones de tiempo especificadas. Si los algoritmos asociados son complejos, se pueden requerir cambios en

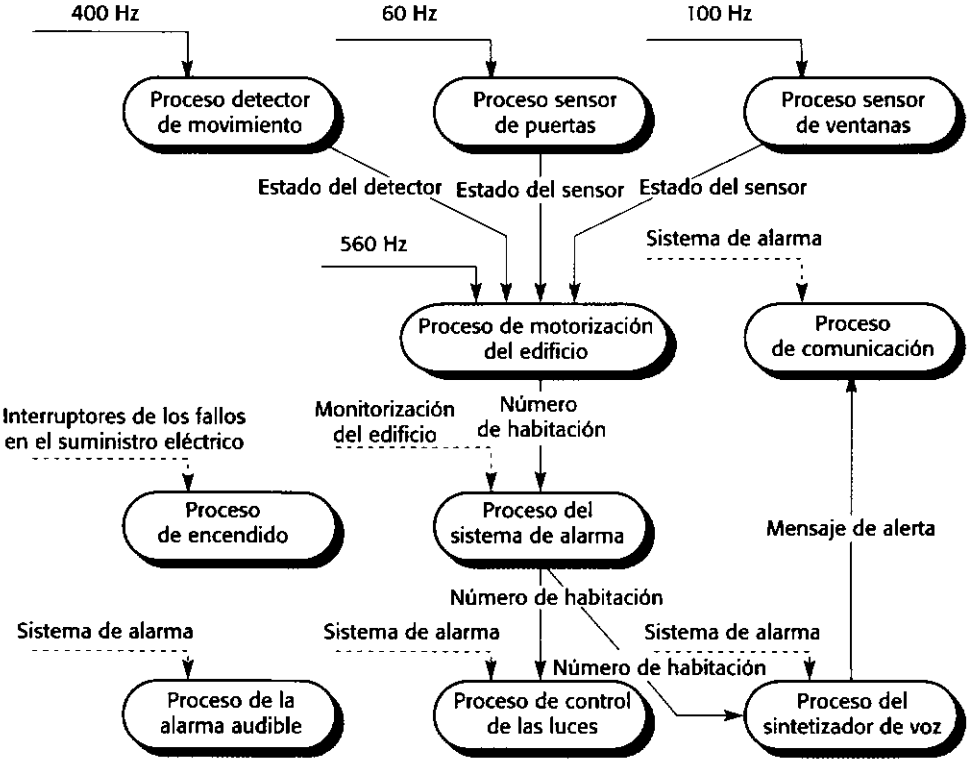


Figura 15.8 Arquitectura de procesos del sistema de alarma antirrobo.

las restricciones de tiempo. Sin embargo, a menos que se requiera el procesamiento de señales, los algoritmos de sistemas de tiempo real son a menudo bastante sencillos. Éstos solamente pueden requerir la comprobación de una posición de memoria, realizar algunos cálculos sencillos o emitir una señal. Como puede verse en la Figura 15.9, el procesamiento requerido en el sistema de alarma antirrobo sigue este sencillo modelo.

```
// Ver http://www.software-engin.com/ para completar el código Java de este ejemplo
class BuildingMonitor extends Thread {
   BuildingSensor win, door, move ;
   Siren     siren = new Siren () ;
   Lights    lights = new Lights () ;
   Synthesizer synthesizer = new Synthesizer () ;
   DoorSensors doors = new DoorSensors (30) ;
   WindowSensors   windows = new WindowSensors (50) ;
   MovementSensors movements = new MovementSensors (200) ;
   PowerMonitor pm = new PowerMonitor () ;
   BuildingMonitor()
   {
      // inicializa todos los sensores e inicia los procesos
      siren.start () ; lights.start () ;
      synthesizer.start () ; windows.start () ;
      doors.start () ; movements.start () ; pm.start () ;
   {}
   {public void run ()
   {{
      int room = 0 ;
      while (true)
      {
         // consulta los sensores de movimiento al menos dos veces por segundo (400 Hz)
         move = movements.getVal () ;
         // consulta los sensores de las ventanas al menos dos veces/segundo (100 Hz)
         win = windows.getVal () ;
         // consulta los sensores de las puertas al menos dos veces por segundo (60 Hz)
         door = doors.getVal () ;
         if (move.sensorVal == 1 | door.sensorVal == 1 | win.sensorVal == 1)
            {
               // un sensor ha detectado un intruso
               if (move.sensorVal == 1)    room = move.room ;
               if (door.sensorVal == 1)    room = door.room ;
               if (win.sensorVal == 1 )    room = win.room ;
               lights.on (room) ; siren.on () ; synthesizer.on (room) ;
               break ;
            }
      }
      lights.shutdown () ; siren.shutdown () ; synthesizer.shutdown () ;
      windows.shutdown () ; doors.shutdown () ; movements.shutdown () ;
   } // run
} //BuildingMonitor
```

Figura 15.9
Implementación Java del proceso BuildingMonitor.

El paso final en el proceso de diseño es el diseño de un sistema de planificación que asegure que un proceso siempre será planificado para cumplir con sus plazos de tiempo. En este ejemplo, los plazos de tiempo no son ajustados. Las prioridades de los procesos deberían organizarse para que todos los procesos que consultan sensores tengan la misma prioridad. El proceso para manejar un fallo en el suministro eléctrico debería ser un proceso de nivel de interrupción con una prioridad más alta. La prioridad de los procesos que gestiona el sistema de alarma debería ser la misma que la de los procesos de los sensores.

El sistema de alarma antirrobo es un sistema de monitorización en vez de un sistema de control, puesto que no incluye actuadores que se vean directamente afectados por los valores del sensor. Un ejemplo de un sistema de control podría ser un sistema de control de la calefacción de un edificio. Este sistema monitoriza los sensores de temperatura en diferentes estancias del edificio y apaga y enciende una unidad de calefacción dependiendo de la temperatura actual y de la temperatura fijada en el termostato de dicha estancia. El termostato también controla la activación del generador de calor del sistema.

La arquitectura de los procesos de este sistema se muestra en la Figura 15.10. Está claro que su forma general es similar al sistema de alarma antirrobo. Se deja al lector el desarrollo del diseño más detallado de este sistema.

15.4 Sistemas de adquisición de datos

Los sistemas de adquisición de datos recogen datos de sensores para su posterior procesamiento y análisis. Estos sistemas se utilizan en circunstancias en las que los sensores han recogido grandes cantidades de datos del entorno del sistema y no es necesario procesar los datos recopilados en tiempo real. Los sistemas de adquisición de datos se usan normalmente en experimentos científicos y sistemas de control de procesos en los que los procesos físicos, tales como una reacción química, ocurren muy rápidamente.

En los sistemas de adquisición de datos, los sensores pueden estar generando datos muy rápidamente, y el problema principal es asegurar que una lectura del sensor es recogida antes

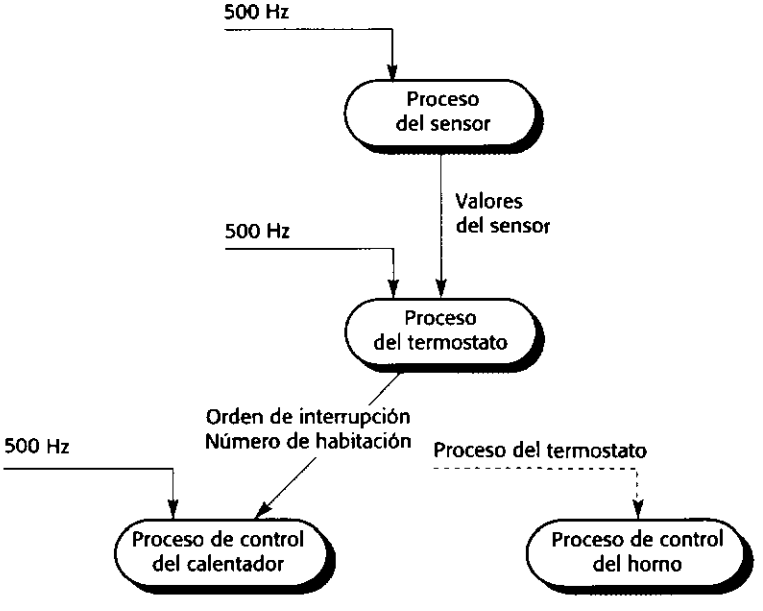


Figura 15.10 Arquitectura de procesos de un sistema de control de temperatura.

de que cambie el valor del sensor. Esto da lugar a una arquitectura genérica tal y como se muestra en la Figura 15.11. La característica fundamental de la arquitectura de los sistemas de adquisición de datos es que cada grupo de sensores tiene tres procesos asociados: el proceso del sensor que interactúa con el sensor y convierte datos analógicos a valores digitales si es necesario, un proceso búfer, y un proceso que consume los datos y realiza un procesamiento adicional.

Por supuesto, los sensores pueden ser de diferentes tipos, y el número de sensores de un grupo depende de la velocidad a la que lleguen los datos desde el entorno. En la Figura 15.11 se muestran dos grupos de sensores, **s1-s3** y **s4-s6**. También se muestra, en la parte de la derecha, un proceso adicional que visualiza los datos del sensor. La mayoría de los sistemas de adquisición de datos incluyen procesos de visualización e informes que reúnen los datos recogidos y realizan un procesamiento adicional.

Como ejemplo de un sistema de adquisición de datos, considere el modelo de sistema mostrado en la Figura 15.12. Éste representa un sistema que recoge datos desde sensores que monitorizan el flujo de neutrones en un reactor nuclear. Los datos del sensor se colocan en un búfer a partir del cual se extraen y procesan, y el nivel promedio del flujo se visualiza en una pantalla del operador.

Cada sensor tiene un proceso asociado que convierte la entrada analógica del nivel de flujo en una señal digital. Dicho proceso envía este nivel de flujo, con el identificador del sensor, al búfer de datos del sensor. El proceso responsable del procesamiento de los datos toma los datos de este búfer, los procesa y los envía a un proceso de visualización para mostrarlos en una consola del operador.

En sistemas de tiempo real que implican la adquisición y el procesamiento de datos, las velocidades y periodos del proceso de adquisición (el productor) y el proceso de procesamiento (el consumidor) pueden no estar sincronizados. Cuando se requiere un procesamiento significativo, la adquisición de datos puede ser más rápida que el procesamiento de los datos. Si solamente es necesario realizar cálculos sencillos, el procesamiento puede ser más rápido que la adquisición de los datos.

Para suavizar estas diferencias de velocidad, los sistemas de adquisición de datos almacenan los datos de entrada utilizando un búfer circular. El proceso que produce los datos (el pro-

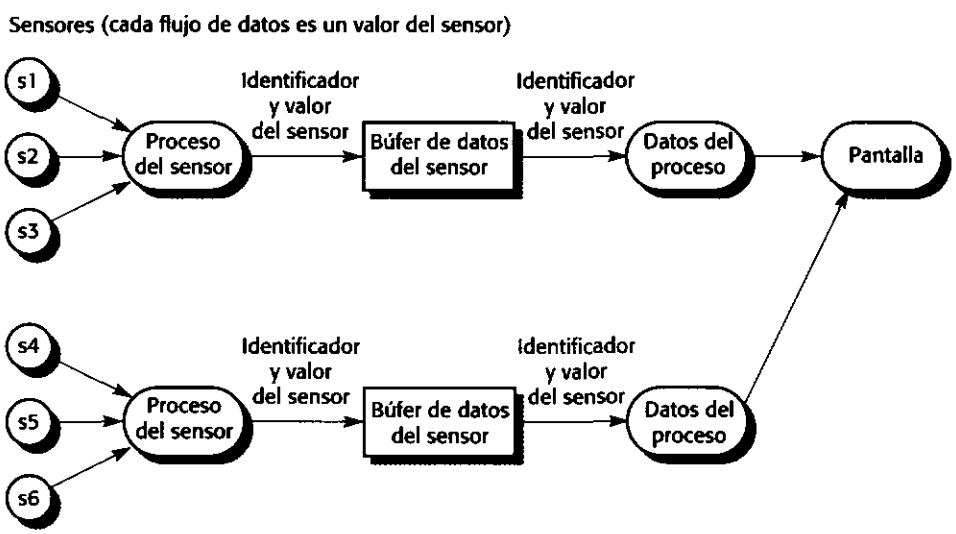


Figura 15.11 Arquitectura genérica de los sistemas de adquisición de datos.

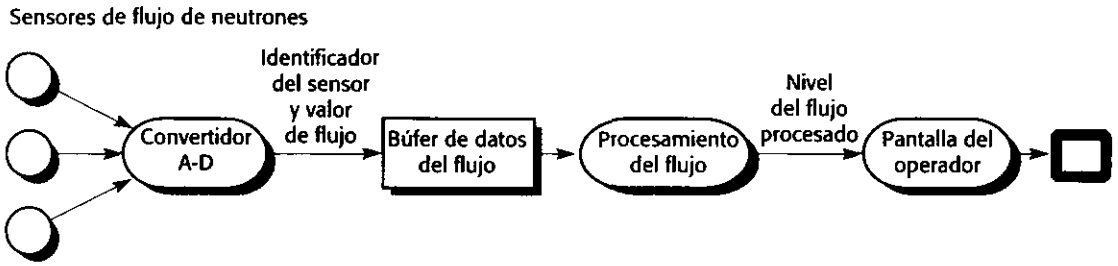


Figura 15.12 Adquisición de datos del flujo de neutrones.

ductor) añade información a este búfer, y el proceso que usa los datos (el consumidor) coge la información del búfer (Figura 15.13).

Obviamente, se debe implementar la exclusión mutua para impedir que los procesos productor y consumidor accedan al mismo elemento del búfer al mismo tiempo. El sistema también debe asegurar que el productor no intente añadir información a un búfer lleno y que el consumidor no extraiga información de un búfer vacío.

En la Figura 15.14, se muestra una posible implementación del búfer de datos como un objeto Java. Los valores en el búfer son del tipo **SensorRecord**, y hay dos operaciones definidas denominadas **get** y **put**. La operación **get** toma un elemento del búfer y la operación **put** añade un elemento al búfer. El constructor del búfer fija el tamaño cuando se declaran los objetos del tipo **CircularBuffer**.

El modificador **synchronized** asociado con los médotos **get** y **put** indica que estos métodos no deberían ejecutarse al mismo tiempo. Cuando se llama a uno de estos métodos, el sistema en tiempo de ejecución obtiene un candado sobre la instancia del objeto para asegurar que el otro método no puede cambiar la misma entrada en el búfer. Las llamadas de los métodos **wait** y **notify** se utilizan para asegurar que no pueden añadirse elementos en un búfer lleno o extraer elemento de un búfer vacío. El método **wait** hace que el hilo de ejecución que realiza dicha llamada se suspenda a sí mismo hasta que otro hilo le comunique que deje de esperar. Esto se realiza llamando al método **notify**. Cuando se realiza una llamada a **wait**, se libera el candado del objeto protegido. El método **notify** despierta a uno de los hilos que está esperando y hace que continúe con su ejecución.

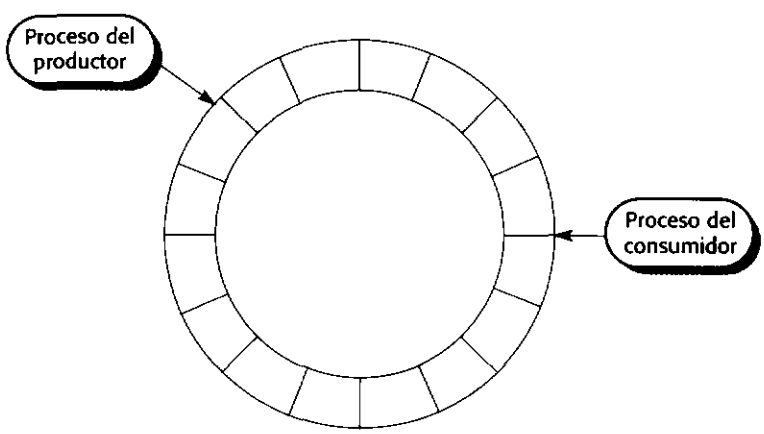


Figura 15.13 Un búfer circular para la adquisición de datos.

```
class CircularBuffer
{
    int bufsize ;
    SensorRecord [] store ;
    int numberOfEntries = 0 ;
    int front = 0, back = 0 ;

    CircularBuffer (int n) {
        bufsize = n ;
        store = new SensorRecord [bufsize] ;
    } // CircularBuffer

    synchronized void put (SensorRecord rec ) throws InterruptedException
    {
        if ( numberOfEntries == bufsize)
            wait () ;
        store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
        back = back + 1 ;
        if (back == bufsize)
            back = 0 ;
        numberOfEntries = numberOfEntries + 1 ;
        notify () ;
    } // put

    synchronized SensorRecord get () throws InterruptedException
    {
        SensorRecord result = new SensorRecord (-1, -1) ;
        if (numberOfEntries == 0)
            wait () ;
        result = store [front] ;
        front = front + 1 ;
        if (front == bufsize)
            front = 0 ;
        numberOfEntries = numberOfEntries-1 ;
        notify () ;
        return result ;
    } // get
} // CircularBuffer
```

Figura 15.14 Una implementación Java de un búfer circular.

PUNTOS CLAVE

- **Un sistema de tiempo real es un sistema software que debe responder a eventos en tiempo real. Su corrección no sólo depende de los resultados que produce sino también del momento en el que se producen dichos resultados.**

- **Un modelo general para la arquitectura de sistemas de tiempo real implica el asociar un proceso con cada clase de dispositivo sensor y actuador. También se requieren procesos adicionales de coordinación.**
- **El diseño arquitectónico de un sistema de tiempo real implica normalmente la organización del sistema como un conjunto de procesos concurrentes que interactúan.**
- **Un sistema operativo de tiempo real es responsable del proceso y la gestión de los recursos. Siempre incluye un planificador, que es el componente responsable de decidir qué proceso debería seleccionarse para su ejecución. Las decisiones de planificación se realizan utilizando las prioridades de los procesos.**
- **Los sistemas de monitorización y control consultan periódicamente un conjunto de sensores que captan información del entorno del sistema. Éstos llevan a cabo acciones, dependiendo de las lecturas de los sensores, y envían órdenes a los actuadores.**
- **Los sistemas de adquisición de datos se organizan normalmente según un modelo productor-consumidor. El proceso productor coloca los datos en un búfer circular, desde donde son consumidos por el proceso consumidor. El búfer también se implementa como un proceso para eliminar los conflictos entre el productor y el consumidor.**

LECTURAS ADICIONALES

Software Engineering for Real-Time Systems. Escrito desde el punto de vista de una ingeniería en lugar de una perspectiva de ciencia de la computación, este libro es una buena guía práctica para la ingeniería de sistemas de tiempo real. Estudia mejor las cuestiones hardware que el libro de Burns y Wellings, por lo que es un complemento excelente a éste. (J. Cooling, 2003, Addison-Wesley.)

Real-time Systems and Programming Languages, 3rd edition. Un texto excelente y fácil de entender que proporciona una amplia cobertura de todos los aspectos de los sistemas de tiempo real. (A. Burns y A. Wellings, 2001, Addison-Wesley.)

Doing Hard Time: Developing Real-Time Systems with UML, Objects Frameworks and Patterns. Este libro explica cómo pueden utilizarse las técnicas orientadas a objetos en el diseño de sistemas de tiempo real. Puesto que la velocidad del hardware cada vez es mayor, este libro se está convirtiendo en una aproximación cada vez más viable al diseño de sistemas de tiempo real. (B. P. Douglass, 1999, Addison-Wesley.)

EJERCICIOS

15.1 Utilizando ejemplos, explique por qué los sistemas de tiempo real tienen que implementarse normalmente utilizando procesos concurrentes.

15.2 Explique por qué una aproximación orientada a objetos para el desarrollo del software puede no ser adecuada para sistemas de tiempo real.

15.3 Dibuje modelos de máquina de estados del software de control para los siguientes sistemas:

- Una lavadora automática que tiene diferentes programas para distintos tipos de ropa.
- El software para un reproductor de discos compactos.

- Un contestador automático de teléfono que graba los mensajes entrantes y visualiza el número de mensajes aceptados en una pantalla de cristal líquido. El sistema debería permitir al propietario del teléfono descolgar, teclear una secuencia de números (identificados como tonos) y reproducir los mensajes grabados en el contestador.
- Una máquina de bebidas que puede dispensar café con y sin leche y azúcar. El usuario inserta una moneda y hace su selección presionando un botón de la máquina. Esto provoca la salida de una taza con café en polvo. El usuario coloca esta taza bajo un grifo, presiona otro botón y se suministra agua caliente.

15.4 Usando las técnicas de diseño de sistemas de tiempo real estudiadas en este capítulo, rediseñe el sistema de recogida de datos de la estación meteorológica tratada en el Capítulo 14 como un sistema de estímulo-respuesta.

15.5 Diseñe una arquitectura de procesos para un sistema de monitorización ambiental que recoge datos de un conjunto de sensores de la calidad del aire situados en varios puntos de una ciudad. Hay 5.000 sensores organizados en 100 barrios. Cada sensor debe ser consultado cuatro veces por segundo. Cuando más del 30% de los sensores en un barrio particular indique que la calidad del aire está por debajo de un nivel aceptable, se activan luces locales de advertencia. Todos los sensores devuelven las lecturas a una computadora central, que genera informes cada 15 minutos sobre la calidad del aire en la ciudad.

15.6 Comente las ventajas e inconvenientes de Java como lenguaje de programación para sistemas de tiempo real. ¿En qué medida los problemas de la programación de tiempo real con Java desaparecerán cuando se utilicen procesadores más rápidos?

15.7 Un sistema de protección de trenes frena automáticamente el tren si el límite de velocidad se excede en un tramo de vía o si el tren entra en un tramo de vía señalizado con una luz roja (es decir, no se debe entrar en ese tramo de vía). Los detalles se muestran en la Figura 15.15. Identifique los estímulos que debe procesar el sistema de control de a bordo del tren y las respuestas asociadas a estos estímulos.

15.8 Sugiera una posible arquitectura de procesos para este sistema. Documente esta arquitectura de procesos utilizando el esquema mostrado en la Figura 15.8, indicando claramente si los estímulos son periódicos o aperiódicos.

15.9 Si un proceso periódico en el sistema de a bordo de protección del tren se utiliza para recoger datos de un transmisor situado en la vía, ¿con qué periodicidad debe planificarse para asegurar que el sistema garantiza la recogida de información del transmisor? Explique la respuesta.

15.10 A usted se le ha solicitado trabajar en un proyecto de desarrollo de tiempo real para una aplicación militar, pero no tiene experiencia previa en proyectos de este dominio. Comente lo que, como ingeniero de software profesional, debería hacer antes de comenzar a trabajar en el proyecto.

- El sistema adquiere información sobre el límite de velocidad de un tramo desde un transmisor situado en la vía, que continuamente emite el identificador del tramo y su límite de velocidad. El mismo transmisor también emite información sobre el estado de la señal que controla ese tramo de vía. El tiempo requerido para emitir el tramo de vía y la información de la señal es de 50 milisegundos.
- El tren puede recibir información desde el transmisor situado en la vía cuando esté a menos de 10 metros del transmisor.
- La velocidad máxima del tren es de 180 km/h.
- Los sensores del tren proporcionan información sobre la velocidad actual del tren (actualizada cada 250 milisegundos) y del estado del freno del tren (actualizado cada 100 milisegundos).
- Si la velocidad del tren excede de la velocidad límite del tramo actual en más de 5 km/h, suena una alarma en la cabina del conductor. Si la velocidad del tren excede de la velocidad límite del tramo actual en más de 10 km/h, los frenos del tren se accionan automáticamente hasta que la velocidad esté dentro de los límites permitidos. Los frenos del tren deberían accionarse antes de 100 milisegundos desde que se detecta la velocidad excesiva del tren.
- Si el tren entra en un tramo de vía señalizado con una luz roja, el sistema de protección del tren acciona los frenos y reduce la velocidad a cero. Los frenos del tren deberían accionarse antes de 100 milisegundos después de que se reciba la señal de luz roja.
- El sistema continuamente actualiza una pantalla de estado en la cabina del conductor.

Figura 15.15 Descripción del sistema de protección de trenes.

16 Diseño de interfaces de usuario

Objetivos

El objetivo de este capítulo es introducir algunos aspectos de diseño de las interfaces de usuario que son importantes para los ingenieros de software. Cuando haya leído este capítulo:

- comprenderá varios principios de diseño de las interfaces de usuario;
- habrá sido introducido a varios estilos de interacción y entenderá cuándo éstos son los más apropiados;
- comprenderá cuándo utilizar presentaciones gráficas y textuales de la información;
- conocerá los aspectos implicados en las principales actividades en el proceso de diseño de las interfaces de usuario;
- comprenderá los atributos de usabilidad y habrá sido introducido a los diversos enfoques de evaluación de interfaces.

Contenidos

El diseño de sistemas informáticos abarca varias actividades que van desde el diseño de hardware hasta el de la interfaz de usuario. Aunque muchos especialistas a menudo trabajan en el diseño de hardware y en el diseño gráfico de páginas web, normalmente sólo las organizaciones grandes emplean diseñadores especialistas de interfaces para sus aplicaciones software. Por tanto, los ingenieros de software a menudo deben tomar la responsabilidad de diseñar la interfaz de usuario, así como del diseño del software que implementa esa interfaz.

Aun cuando los diseñadores y programadores de software son usuarios competentes en la tecnología utilizada en la implementación de las interfaces, como las clases Swing de Java (Elliott *et al.*, 2002) o XHTML (Musciano y Kennedy, 2002), las interfaces de usuario que desarrollan a menudo son poco atractivas e inapropiadas para sus usuarios objetivo. Se analizará, por lo tanto, el proceso de diseño de interfaces de usuario en lugar del software que implementa estas interfaces. Debido a las limitaciones de espacio, sólo se considerarán las interfaces gráficas de usuario. No se tratarán las interfaces que requieren una forma especial (aunque quizá muy simple) de visualización como las de telefonía móvil, reproductores de DVD, televisores, copiadoras y máquinas de fax. Naturalmente, aquí sólo se puede dar una introducción al tema, por lo que para obtener mayor información sobre el diseño de interfaces de usuario se recomiendan los textos de Dix y otros (Dix *et al.*, 2004), Weiss (Weiss, 2002) y Shneiderman (Shneiderman, 1998).

Un diseño cuidadoso de la interfaz de usuario es parte fundamental del proceso de diseño general del software. Si un sistema software debe alcanzar su potencial máximo, es fundamental que su interfaz de usuario sea diseñada para ajustarse a las habilidades, experiencia y expectativas de sus usuarios previstos. Un buen diseño de la interfaz de usuario es crítico para la confiabilidad del sistema. Muchos de los llamados «errores de usuario» son causados por el hecho de que las interfaces de usuario no consideran las habilidades de los usuarios reales y su entorno de trabajo. Una interfaz de usuario mal diseñada significa que los usuarios probablemente no podrán acceder a algunas características del sistema, cometerán errores y sentirán que el sistema les dificulta en vez de ayudarlos a conseguir cualquier objetivo para el que utilizan el sistema.

Cuando se toman decisiones en el diseño de las interfaces de usuario, deben tenerse en cuenta las capacidades físicas y mentales de las personas que utilizarán el software. Las limitaciones de espacio no permite tratar aquí en detalle los factores humanos, pero algunos factores importantes que deben considerarse son los siguientes:

1. Las personas tienen una memoria limitada a corto plazo: podemos recordar instantáneamente alrededor de siete elementos de información (Miller, 1957). Por lo tanto, si a los usuarios se les presenta demasiada información al mismo tiempo, es posible que no puedan asimilarla.
2. Todos cometemos errores, especialmente cuando tenemos que manejar demasiada información o estamos estresados. Cuando los sistemas fallan y emiten mensajes de aviso y alarmas, a menudo aumentan el estrés de los usuarios, incrementando así la posibilidad de que cometan errores.
3. Poseemos un amplio rango de capacidades físicas. Unas personas ven y escuchan mejor que otras, otras son daltónicas, y otras son mejores en manipulaciones físicas. No se debe diseñar para las propias capacidades y suponer que todos los otros usuarios serán capaces de adaptarse.
4. Tenemos diferentes preferencias de interacción. A algunas personas les gusta trabajar con imágenes, a otras con texto. La manipulación directa es natural para algunas personas, pero otras prefieren un estilo de interacción basado en emitir comandos al sistema.

Estos factores humanos son la base para los principios de diseño que se muestran en la Figura 16.1. Estos principios generales se aplican a todos los diseños de interfaces de usuario y normalmente se deben instanciar como directrices de diseño más detalladas para organizaciones o tipos de sistema específicos. Los principios de diseño de las interfaces de usuario son tratados con mayor detalle por Dix y otros (Dix *et al.*, 2004). Shneiderman (Shneiderman, 1998) da una lista más amplia de directrices más específicas para el diseño de las interfaces de usuario.

El principio de *familiaridad del usuario* sugiere que los usuarios no deben ser obligados a adaptarse a una interfaz sólo porque sea conveniente implementarla. La interfaz debe utilizar términos familiares para los usuarios, y los objetos que el sistema manipula deben estar directamente relacionados con el entorno de trabajo del usuario. Por ejemplo, si un sistema se diseña para ser utilizado por controladores del tráfico aéreo, los objetos deben ser aviones, trayectorias de vuelo, aerofaros, etcétera. Las operaciones asociadas podrían ser incrementar o reducir la velocidad del avión, ajustar la posición del avión y cambiar de altura. La implementación subyacente de la interfaz en lo que se refiere a archivos y estructuras de datos se debe ocultar al usuario final.

El principio de *uniformidad de la interfaz de usuario* significa que los comandos y menús del sistema deben tener el mismo formato, los parámetros deben pasarse a todos los comandos de la misma forma, y la puntuación de los comandos debe ser similar. Las interfaces uniformes reducen el tiempo de aprendizaje del usuario. Por lo tanto, el conocimiento aprendido en un comando o aplicación es aplicable en otras partes del sistema o en aplicaciones relacionadas.

La uniformidad de la interfaz a lo largo de las aplicaciones también es importante. En lo posible, los comandos con significados similares en aplicaciones diferentes se deben expresar de la misma forma. A menudo los errores se originan cuando el mismo comando del teclado, como «Control+b», significa cosas diferentes en sistemas distintos. Por ejemplo, en el procesador de textos que utilizo normalmente, «Control+b» significa poner en negrita el texto, pero en los programas gráficos que utilizo para dibujar diagramas, «Control+b» significa poner el objeto seleccionado detrás de otro objeto. Yo cometo errores cuando los utilizo al mismo tiempo y a menudo trato de poner en negrita el texto en un diagrama utilizando esta combinación de teclas. Me confundo entonces cuando el texto desaparece detrás del objeto que lo encierra. Normalmente, se puede evitar este tipo de errores si se siguen los métodos abreviados para las teclas de comandos definidos por el sistema operativo que utiliza.

Principio	Descripción
Familiaridad del usuario	La interfaz debe utilizar términos y conceptos obtenidos de la experiencia de las personas que más utilizan el sistema.
Uniformidad	Siempre que sea posible, la interfaz debe ser uniforme en el sentido de que las operaciones comparables se activen de la misma forma.
Mínima sorpresa	El comportamiento del sistema no debe provocar sorpresa a los usuarios.
Recuperabilidad	La interfaz debe incluir mecanismos para permitir a los usuarios recuperarse de los errores.
Guía de usuario	Cuando ocurran errores, la interfaz debe proporcionar retroalimentación significativa y características de ayuda sensible al contexto.
Diversidad de usuarios	La interfaz debe proporcionar características de interacción apropiadas para los diferentes tipos de usuarios del sistema.

Figura 16.1 Principios de diseño de las interfaces de usuario.

Este nivel de uniformidad es de bajo nivel. Los diseñadores de interfaces siempre deben tratar de conseguir este nivel en una interfaz de usuario. Algunas veces, la uniformidad de nivel alto también es deseable. Por ejemplo, puede ser conveniente llevar a cabo las mismas operaciones (como imprimir, copiar, etcétera) sobre todos los tipos de entidades del sistema. Sin embargo, Grudin (Grudin, 1989) señala que la uniformidad total no siempre es posible o deseable. Puede ser razonable implementar el borrado de un escritorio arrastrando las entidades a un cubo de basura, pero sería incómodo borrar el texto en un procesador de textos de esta forma.

Desgraciadamente, los principios de familiaridad del usuario y uniformidad a veces son contradictorios. Idealmente, las aplicaciones con características comunes deberían utilizar siempre los mismos comandos para acceder a estas características. Sin embargo, esto puede chocar con los hábitos del usuario cuando los sistemas se diseñan para apoyar a un tipo de usuario en particular, como los diseñadores gráficos. Estos usuarios pueden tener que desarrollar sus propios estilos de interacciones, terminología y convenciones de funcionamiento. Éstas pueden estar en desacuerdo con los «estándares» de interacción que son apropiados para aplicaciones más generales como los procesadores de textos.

El principio de *mínima sorpresa* es apropiado debido a que las personas se irritan demasiado cuando el sistema se comporta de forma inesperada. Cuando se utiliza un sistema, los usuarios construyen un modelo mental de la forma en que trabaja dicho sistema. Si una acción en algún contexto provoca un tipo de cambio particular, es razonable esperar que la misma acción en un contexto diferente cause un cambio comparable. Si sucede algo completamente diferente, el usuario se sorprende y confunde. Por lo tanto, los diseñadores de interfaces deben intentar asegurar que las acciones comparables tengan efectos comparables.

Las sorpresas en las interfaces de usuario a menudo se deben al hecho de que en muchas interfaces existen varios modos de trabajo (por ejemplo, el modo vista y el modo edición), y el efecto de un comando es diferente dependiendo del modo. Es muy importante que, al diseñar una interfaz, se incluya un indicador visual que muestre al usuario el modo actual.

El principio de *recuperabilidad* es importante debido a que los usuarios inevitablemente cometen errores cuando utilizan un sistema. El diseño de la interfaz puede minimizar estos errores (por ejemplo, los errores de teclado se evitan si se utilizan menús), pero los errores nunca pueden eliminarse completamente. Por consiguiente, se deben incluir recursos que permitan a los usuarios recuperarse de sus errores. Éstos pueden ser de tres tipos:

1. *Confirmación de acciones destructivas.* Si un usuario lleva a cabo una acción que es potencialmente destructiva, el sistema debería pedirle que confirme que esto es realmente lo que desea antes de destruir cualquier información.
2. *Proporcionar un recurso para deshacer.* El recurso deshacer restablece el sistema al estado previo antes de que ocurriera la acción. Son útiles varios niveles de este recurso debido a que los usuarios no siempre reconocen inmediatamente que han cometido un error.
3. *Generar puntos de control.* La generación de puntos de control implica grabar el estado de un sistema en intervalos periódicos y permitir que el sistema se restaure desde el último punto de control. De esta forma, cuando se produce un error, los usuarios pueden retroceder a un estado previo y empezar de nuevo. En la actualidad, muchos sistemas incluyen la generación de puntos de control para tratar los fallos del sistema pero, paradójicamente, no permiten a los usuarios del sistema utilizarlos para recuperarse de sus propios errores.

Un principio relacionado es el de *asistencia al usuario*. Las interfaces deben proporcionar asistencia al usuario o características de ayuda. Éstas se deben integrar en el sistema y proporcionar diferentes niveles de ayuda y asesoramiento. Los niveles deben variar desde la in-

formación básica para iniciarse con el sistema hasta una descripción completa de las características del sistema. Los sistemas de ayuda se deben estructurar de forma que cuando el usuario requiera ayuda no se sienta saturado con la información.

El principio de *diversidad de usuarios* señala que, para muchos sistemas interactivos, pueden existir diferentes tipos de usuarios. Algunos son usuarios casuales y tienen interacción con el sistema ocasionalmente, mientras que otros son usuarios potenciales que utilizan el sistema durante varias horas todos los días. Los usuarios casuales necesitan interfaces que los guíen, pero los usuarios potenciales requieren métodos abreviados de forma que puedan interactuar con el sistema tan rápido como sea posible. Además, los usuarios pueden tener impedimentos de varios tipos y, si es posible, las interfaces deben poder adaptarse para hacer frente a esto. Por lo tanto, se podría incluir recursos para mostrar diferentes tamaños de texto, reemplazar el sonido con texto, permitir modificar el tamaño de los botones, etcétera. Esto refleja la noción de Diseño Universal (UD) (Preiser y Ostoff, 2001), un principio de diseño cuyo objetivo es evitar excluir usuarios debido a elecciones de diseño irreflexivas.

El principio de reconocimiento de la diversidad de usuarios puede estar en pugna con los otros principios de diseño de las interfaces, ya que algunos usuarios pueden preferir una interacción muy rápida sobre, por ejemplo, la uniformidad de la interfaz. De forma similar, el nivel de ayuda requerido puede ser radicalmente diferente para distintos usuarios, y puede ser imposible desarrollar ayudas adecuadas para todos los tipos de usuario. Por lo tanto, se debe llegar a acuerdos para hacer compatibles las necesidades de estos usuarios.

16.1 Asuntos de diseño

Antes de abordar el proceso de diseño de la interfaz de usuario, se tratan algunos asuntos generales de diseño que tienen que ser considerados por los diseñadores de interfaces de usuario. Fundamentalmente, el diseñador de una interfaz de usuario se plantea dos cuestiones clave:

1. ¿Cómo debe interactuar el usuario con el sistema informático?
2. ¿Cómo se debe presentar la información del sistema informático al usuario?

Una interfaz de usuario coherente debe integrar la interacción del usuario y la presentación de la información. Esto puede ser difícil debido a que los diseñadores tienen que encontrar un término medio entre los estilos más adecuados de interacción y presentación para la aplicación, la formación y experiencia de los usuarios del sistema, y el equipo disponible.

16.1.1 Interacción del usuario

La interacción del usuario significa emitir comandos y datos asociados al sistema informático. En las primeras computadoras, la única forma de hacer esto era a través de una interfaz de línea de comandos, y se utilizaba un lenguaje de propósito específico para comunicarse con la máquina. Sin embargo, este enfoque se orientó a los usuarios expertos y actualmente se han desarrollado varios enfoques que son más fáciles de utilizar. Shneiderman (Shneiderman, 1998) ha clasificado estas formas de interacción en cinco estilos principales:

1. *Manipulación directa.* El usuario interactúa directamente con los objetos de la pantalla. La manipulación directa normalmente implica un dispositivo apuntador (un ratón, un lápiz óptico, un *trackball* o, en una pantalla táctil, un dedo) que indica el objeto a manipular y la acción, la cual especifica lo que se debe hacer con ese objeto. Por ejem-

plo, para borrar un archivo, se puede hacer clic en un icono que represente a ese archivo y arrastrarlo a un icono de un cubo de basura.

2. *Selección de menús.* El usuario selecciona un comando de una lista de posibilidades (un menú). También puede seleccionar otro objeto de la pantalla por manipulación directa, y el comando actúa sobre él. En este enfoque, para borrar un archivo, seleccionaría el icono del archivo y después el comando de borrado.
3. *Rellenado de formularios.* El usuario rellena los campos de un formulario. Algunos campos pueden llevar menús asociados, y el formulario puede tener «botones» de acción que, cuando se presionan, hacen que se inicie alguna acción. Normalmente no utilizará este enfoque para implementar la interfaz de operaciones como el borrado de archivos. Hacer esto implicaría introducir el nombre del archivo en el formulario y después «presionar» un botón de borrar.
4. *Lenguaje de comandos.* El usuario emite un comando especial y los parámetros asociados para indicar al sistema qué hacer. Para borrar un archivo, se teclearía un comando de borrado con el del archivo como parámetro.
5. *Lenguaje natural.* El usuario emite un comando en lenguaje natural. Normalmente esto es un *front-end* para un lenguaje de comandos; el lenguaje natural se analiza y traduce a comandos del sistema. Para borrar un archivo, se teclearía «borrar el archivo xxx».

Estilo de interacción	Principales ventajas	Principales desventajas	Ejemplos de aplicación
Manipulación directa	Interacción rápida e intuitiva. Fácil de aprender	Puede ser difícil de implementar. Sólo es adecuada donde existe una metáfora visual para las tareas y objetos	Videojuegos. Sistemas CAD
Selección de menús	Evita errores del usuario. Se requiere teclear poco	Lenta para usuarios experimentados. Puede llegar a ser compleja si existen muchas opciones en el menú	La mayoría de los sistemas de propósito general
Rellenado de formularios	Introducción de datos sencilla	Ocupa mucho espacio en la pantalla. Causa problemas cuando las opciones del usuario no se ajustan a los campos del formulario	Control de stock. Procesamiento de préstamos personales
Lenguaje de comandos	Poderoso y flexible	Difícil de aprender. Gestión pobre de errores	Sistemas operativos. Sistemas de comandos y control
Lenguaje natural	Accesible a usuarios casuales. Fácil de ampliar	Se requiere teclear más. Los sistemas de comprensión de lenguaje natural no son fiables	Sistemas de recuperación de información

Figura 16.2 Ventajas y desventajas de los estilos de interacción.

Cada uno de estos estilos de interacción tiene ventajas y desventajas y es más adecuado para diferentes tipos de aplicaciones y usuarios (Shneiderman, 1998). La Figura 16.2 muestra las principales ventajas y desventajas de estos estilos y sugiere los tipos de aplicación donde podrían utilizarse.

Por supuesto, estos estilos de interacción se pueden mezclar, utilizando varios estilos en la misma aplicación. Por ejemplo, Microsoft Windows permite la manipulación directa de los iconos que representan a los archivos y carpetas, la selección de comandos basados en menús, y para comandos como los de configuración, los usuarios deben rellenar un formulario de propósito específico que se les muestra.

En principio, es posible separar el estilo de interacción de las entidades subyacentes que se manipulan a través de la interfaz de usuario. Ésta fue la base para el modelo de Seeheim (Pfaff y ten Hagen, 1985) sobre la gestión de interfaces de usuario. En este modelo, se separa la presentación de la información, la gestión del diálogo y la aplicación. En la realidad, éste es un modelo ideal más que un modelo práctico, aunque es ciertamente posible tener interfaces separadas para los diferentes tipos de usuarios (usuarios casuales y usuarios experimentados) que interactúan con el mismo sistema subyacente. Esto se ilustra en la Figura 16.3, la cual muestra una interfaz de lenguaje de comandos y una interfaz gráfica para un sistema operativo como Linux.

Las interfaces de usuario basadas en web se fundan en el soporte proporcionado por HTML o XHTML (el lenguaje de descripción de páginas utilizado por las páginas web) junto con lenguajes como Java, los cuales pueden asociar programas con los componentes de una página. Debido a que normalmente estas interfaces basadas en web son diseñadas por usuarios casuales, la mayoría de ellas utiliza interfaces basadas en formularios. Es posible construir interfaces de manipulación directa en web, pero ésta es una tarea compleja de programación. Además, debido a los diferentes niveles de experiencia de los usuarios de la web y al hecho de que provienen de muchas culturas diferentes, es difícil establecer una metáfora de la interfaz de usuario para la interacción directa que sea universalmente aceptada.

Para ilustrar el diseño de la interacción de usuario basada en web, se presenta el enfoque utilizado en el sistema LIBSYS donde los usuarios pueden acceder a documentos de otras bibliotecas. Existen dos operaciones fundamentales que se necesita soportar:

1. *Búsqueda de documentos*, en la que los usuarios utilizan las funciones de búsqueda para encontrar los documentos que necesitan.
2. *Petición de documentos*, en la que los usuarios solicitan que el documento se descargue en su máquina o servidor local para imprimirlo.

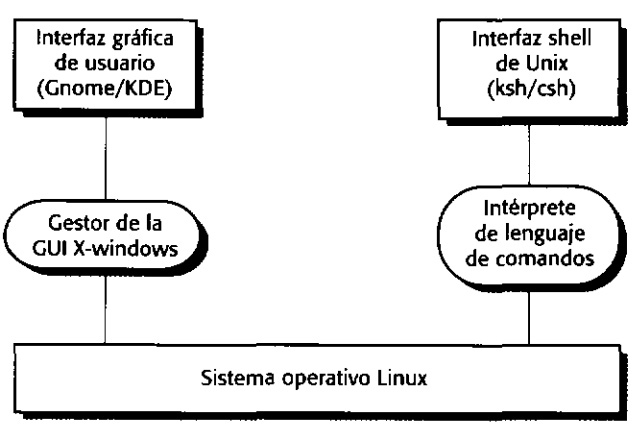


Figura 16.3
Múltiples interfaces de usuario.

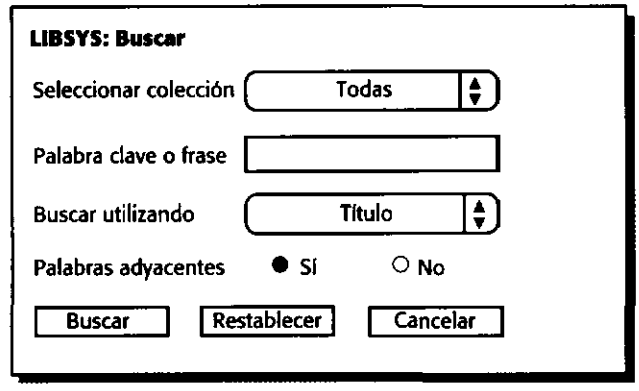


Figura 16.4 Una interfaz basada en formularios para el sistema LIBSYS.

La interfaz de usuario del LIBSYS se implementa utilizando un navegador web, por lo que, dado que los usuarios deben proporcionar información al sistema como el identificador del documento, sus nombres y detalles de autorización, tiene sentido utilizar una interfaz basada en formularios. La Figura 16.4 muestra un posible diseño de la interfaz para la búsqueda de componentes del sistema.

En interfaces basadas en formularios, el usuario proporciona toda la información requerida y después inicia la acción pulsando un botón. Los campos de los formularios pueden ser menús, campos de entrada libre de texto o botones de radio. En el ejemplo del LIBSYS, un usuario selecciona la colección a buscar de un menú de colecciones al que se puede acceder (la opción por defecto es «Todas», que significa buscar todas las colecciones) y teclea la frase de búsqueda en un campo de entrada libre de texto. El usuario elige el campo del archivo de la biblioteca de un menú (la opción por defecto es «Título») y selecciona un botón de radio para indicar si los términos de búsqueda deben ser adyacentes en el archivo.

16.1.2 Presentación de la información

Todos los sistemas interactivos tienen que proporcionar alguna forma de presentar la información a los usuarios. La presentación de la información puede ser simplemente una representación directa de la información de entrada (por ejemplo, texto en un procesador de textos) o presentar la información gráficamente. Una buena pauta de diseño es mantener separado el software requerido para la presentación de la información misma. Separar el sistema de presentación de los datos nos permite cambiar la representación en la pantalla del usuario sin tener que cambiar el sistema de cálculo subyacente. Esto se ilustra en la Figura 16.5.

El enfoque MVC (Figura 16.6), el cual estuvo disponible por primera vez en Smalltalk (Goldberg y Robson, 1983), es una forma efectiva para permitir representaciones múltiples

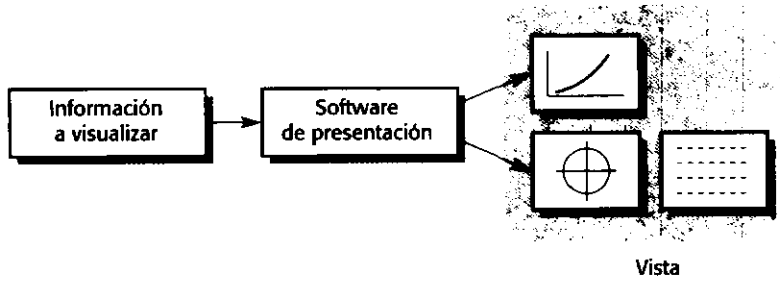


Figura 15.5 Presentación de la información.

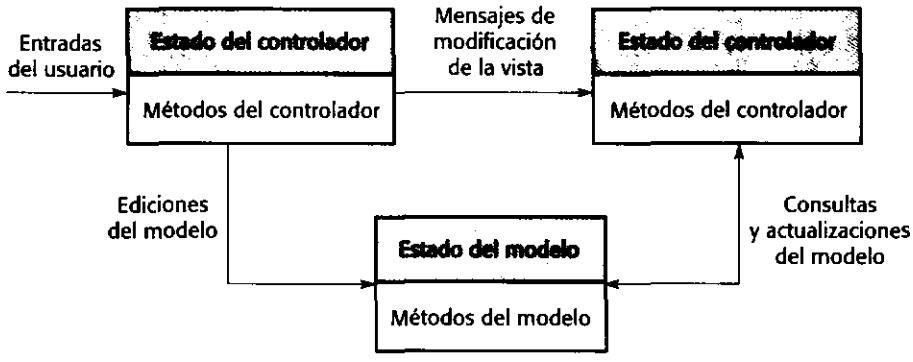


Figura 16.6 El modelo MVC de interacción del usuario.

de datos. Los usuarios pueden interactuar con cada presentación utilizando un estilo apropiado a ésta. Los datos a visualizar se encapsulan en un modelo de objetos. Cada modelo de objetos puede tener asociados varios objetos vista diferentes donde cada vista es una representación de visualización diferente del modelo.

Cada vista tiene un objeto controlador asociado que maneja las entradas del usuario y la interacción de los dispositivos. Por lo tanto, un modelo que representa datos numéricos puede tener una vista que represente los datos como un histograma y una vista que presente los datos como una tabla. El modelo se puede editar cambiando los valores en la tabla o alargando o acortando las barras en el histograma. Esto se trata con mayor detalle en el Capítulo 18, donde se explica cómo puede utilizarse el patrón Observer para implementar el marco de trabajo MVC.

Para encontrar la mejor presentación de la información, se necesita conocer a los usuarios de la información y saber cómo utilizarán el sistema. Cuando se decide cómo presentar la información, deben tenerse presentes las siguientes cuestiones:

1. ¿El usuario está interesado en información precisa o en las relaciones entre los valores de los datos?
2. ¿Con qué frecuencia cambian los valores de la información? ¿Se indicarán de forma inmediata al usuario los cambios en un valor?
3. ¿El usuario debe llevar a cabo alguna acción en respuesta a los cambios de la información?
4. ¿El usuario necesita interactuar con la información visualizada a través de una interfaz de manipulación directa?
5. ¿La información que se va a visualizar es textual o numérica? ¿Son importantes los valores relativos de los elementos de la información?

No se debe suponer que por utilizar gráficos se hacen las vistas más interesantes. Los gráficos ocupan un valioso espacio en la pantalla (una cuestión importante en los dispositivos móviles) y pueden tardar bastante tiempo en descargarse si el usuario está trabajando con una conexión de acceso telefónico lenta.

Dependiendo de la aplicación, la información que no cambia durante una sesión se puede presentar tanto gráfica como textualmente. La presentación textual ocupa menos espacio en la pantalla, pero no se puede leer de un vistazo. Debe distinguirse la información que no cambia de la información dinámica utilizando diferentes estilos de presentación. Por ejemplo, podría presentarse toda la información estática con un tipo de letra o color particular, o podría asociarse con un icono de «información estática».

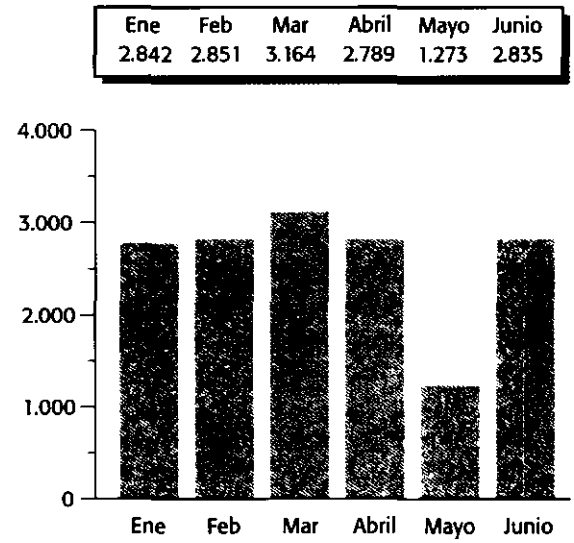


Figura 16.7 Presentaciones alternativas de la información.

Se debe utilizar texto para presentar la información cuando se requiere que ésta sea precisa y cambie de forma relativamente lenta. Si los datos cambian rápidamente o si las relaciones entre los datos más que los valores exactos de los datos son importantes, se debe presentar la información gráficamente.

Por ejemplo, considere un sistema que registra y resume las cifras de venta mensuales de una compañía. La Figura 16.7 ilustra cómo presentar la misma información como texto o en forma gráfica. Normalmente, los gerentes que estudian las cifras de ventas están más interesados en las tendencias o anomalías que en los valores exactos. La presentación gráfica de esta información, como un histograma, hace que las cifras anómalas en marzo y en mayo destaquen sobre las otras. La citada figura también ilustra cómo la presentación textual ocupa menos espacio que la presentación gráfica de la misma información.

En las salas de control o los tableros de mandos como los del salpicadero de un coche, la información que se muestra representa el estado de algún otro sistema (por ejemplo, la altitud de un avión) y está cambiando continuamente. Las vistas digitales que cambian constantemente pueden ser confusas y molestas ya que los lectores no pueden leer y asimilar la información antes de que cambie. Por lo tanto, la información numérica que varía dinámicamente se representa mejor de forma gráfica utilizando una representación analógica. Si es necesario, las vistas gráficas pueden complementarse con una vista digital precisa. En la Figura 16.8 se muestran diferentes formas de presentar información numérica dinámica.

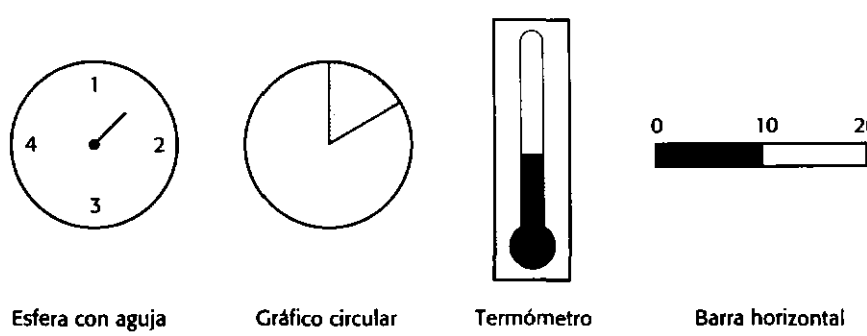


Figura 16.8 Métodos de presentar la información numérica que varía dinámicamente.

Las vistas analógicas continuas dan al usuario una sensación de valor relativo. En la Figura 16.9, los valores de la temperatura y la presión son aproximadamente los mismos. Sin embargo, la vista gráfica muestra que la temperatura está cerca de su valor máximo mientras que la presión no alcanza el 25% de su máximo. Con un valor digital solamente, el usuario necesita conocer los valores máximos y calcular mentalmente el estado relativo de la lectura. En situaciones de estrés, pensar adicionalmente puede conducir a errores humanos que generan problemas, por lo que las vistas pueden mostrar lecturas anormales.

Cuando se tienen que presentar grandes cantidades de información, se pueden utilizar visualizaciones abstractas que vinculen los elementos de los datos relacionados. Esto puede revelar relaciones que no son obvias en los datos sin formato. Se debe ser consciente de las posibilidades de visualización, especialmente cuando la interfaz de usuario del sistema debe representar entidades físicas. He aquí algunos ejemplos de visualizaciones de datos:

1. La información meteorológica, recogida de varias fuentes, se muestra como un mapa meteorológico con isobaras, frentes meteorológicos, etcétera.
2. El estado de una red telefónica se muestra gráficamente como un conjunto vinculado de nodos en un centro de administración de la red.
3. El estado de una planta química se visualiza mostrando las presiones y temperaturas en un conjunto vinculado de depósitos y tuberías.
4. Un modelo de una molécula se muestra y manipula en tres dimensiones utilizando un sistema de realidad virtual.
5. Un conjunto de páginas web se muestra como un árbol hiperbólico (Lamping *et al.*, 1995).

Shneiderman (Shneiderman, 1998) ofrece una buena visión general de los enfoques para la visualización además de identificar las clases de visualización que se pueden utilizar. Éstas incluyen la visualización de datos utilizando presentaciones en dos y tres dimensiones y en forma de árboles o redes. La mayoría de éstas se refieren a la visualización de grandes cantidades de información gestionada en una computadora. Sin embargo, la utilización más común de la visualización en las interfaces de usuario es para representar alguna estructura física, como la estructura molecular de una nueva droga, las conexiones en una red de telecomunicaciones, etcétera. Las presentaciones en tres dimensiones que pueden utilizar equipo de realidad virtual especial son particularmente eficaces para producir visualizaciones. Una forma muy eficaz para interactuar con los datos es la manipulación directa de estas visualizaciones.

Además del estilo de presentación de la información, se debe pensar detenidamente en los colores utilizados en la interfaz. El color puede mejorar las interfaces de usuario ayudando a los usuarios a comprender y manejar la complejidad. Sin embargo, es fácil utilizar el color de forma errónea para crear interfaces visualmente poco atractivas y propensas a errores. Shneiderman da 14 pautas clave para la utilización efectiva del color en las interfaces de usuario. Las más importantes son:

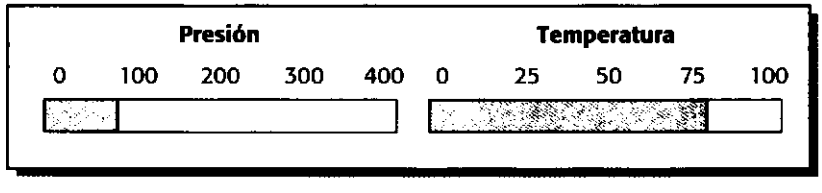


Figura 16.9 Vista de información gráfica que muestra valores relativos.

1. *Limitar el número de colores utilizados y ser conservador en la forma de utilizarlos.* No deben utilizarse más de cuatro o cinco colores diferentes en una ventana y no más de siete en una interfaz del sistema. Si se utilizan demasiados, o si son demasiado vivos, la vista puede ser confusa. Algunos usuarios pueden considerar que las grandes cantidades de colores son molestas y visualmente cansadas. También es posible la confusión en el usuario si los colores no se utilizan de manera uniforme.
2. *Utilizar un cambio de color para mostrar un cambio en el estado del sistema.* Si una vista cambia de color, debe significar que ha ocurrido un evento importante. Así, en un indicador del nivel de combustible, se podría utilizar un cambio de color para indicar una bajada. Resaltar el color es muy importante en las vistas complejas donde se muestran cientos de entidades distintas.
3. *Utilizar el código de colores para apoyar la tarea que los usuarios están tratando de llevar a cabo.* Si los usuarios tienen que identificar instancias anómalas, se deben resaltar estas instancias; si también tienen que descubrir similitudes, se deben resaltar éstas utilizando un color diferente.
4. *Utilizar el código de colores de una forma consciente y uniforme.* Si una parte de un sistema muestra los mensajes de error en rojo (por ejemplo), todas las demás partes deben mostrarlos de igual forma. El rojo no se debe utilizar para nada más. Si se hace, es posible que el usuario interprete la vista en rojo como un mensaje de error.
5. *Ser cuidadoso al utilizar pares de colores.* Debido a la fisiología del ojo, las personas no pueden enfocar el rojo y el azul simultáneamente. La vista cansada es una consecuencia probable de una vista en rojo sobre azul. Otras combinaciones de colores pueden ser también visualmente molestas o difíciles de leer.

En general, se debe utilizar el color para la acción de resaltar, pero no se debe asociar significados con colores particulares. Aproximadamente el 10% de los hombres son daltónicos y pueden malinterpretar el significado. Las percepciones humanas del color son diferentes, y existen convenciones distintas en diferentes profesiones acerca del significado de colores particulares. Los usuarios con conocimientos diferentes inconscientemente pueden interpretar el mismo color de formas distintas. Por ejemplo, para un conductor, el rojo por lo general significa *peligro*. Sin embargo, para un químico, el rojo significa *caliente*.

Además de presentar la información de la aplicación, los sistemas también se comunican con los usuarios a través de mensajes que proporcionan información sobre los errores y el estado del sistema. La primera experiencia de un usuario de un sistema software puede ser cuando el sistema presenta un mensaje de error. Los usuarios inexpertos pueden empezar su trabajo, cometer un error inicial y de forma inmediata tienen que comprender el mensaje de error resultante. Esto puede ser bastante difícil para los ingenieros de software expertos. Es a menudo imposible para los usuarios inexpertos o casuales del sistema. En la Figura 16.10 se muestran los factores que deben tenerse en cuenta al diseñar mensajes del sistema.

Se debe prever la formación y experiencia de los usuarios cuando se diseñan mensajes de error. Por ejemplo, suponga que un usuario del sistema es una enfermera en una sala de cuidados intensivos de un hospital. La observación del paciente se lleva a cabo mediante un sistema informático. Para ver el estado actual del paciente (ritmo cardíaco, temperatura, etcétera), la enfermera selecciona «mostrar» de un menú e introduce el nombre del paciente en un recuadro, como se muestra en la Figura 16.11.

En este caso, vamos a suponer que la enfermera ha escrito incorrectamente el nombre del paciente y ha tecleado «MacDonald» en lugar de «McDonald». El sistema genera un mensa-

Factor	Descripción
Contexto	Donde sea posible, los mensajes generados por el sistema deben reflejar el contexto actual del usuario. En lo posible, el sistema debe ser consciente de lo que está haciendo el usuario y generar mensajes relacionados con su actividad actual.
Experiencia	Al aumentar la familiaridad de los usuarios con el sistema, también se aumenta su molestia por los mensajes largos y «significativos». Sin embargo, los principiantes tienen dificultades en comprender los mensajes cortos y concisos del problema. Se deben proporcionar ambos tipos de mensajes y permitir al usuario controlar la concisión de los mensajes.
Nivel de habilidad	Los mensajes se deben adaptar a las habilidades del usuario, así como a su experiencia. Los mensajes para las diferentes clases de usuario se pueden expresar de diferentes formas dependiendo de la terminología que el lector utilice.
Estilo	Los mensajes deben ser positivos en vez de negativos. Deben estar escritos en modo activo y no en pasivo. No deben ser insultantes o tratar de ser graciosos.
Cultura	En la medida de lo posible, el diseñador de mensajes debe estar familiarizado con la cultura del país donde se vende el sistema. Existen distintas diferencias culturales entre Europa, Asia y América. Un mensaje adecuado en una cultura podría no aceptarse en otra.

Figura 16.10 Factores de diseño en la redacción de mensajes.

je de error. Los mensajes de error siempre deben ser formales, concisos, uniformes y constructivos. No deben ser ofensivos ni tener sonidos asociados u otro tipo de ruidos que pueden desconcertar al usuario. En la medida de lo posible, el mensaje debe sugerir cómo se podría corregir el error. El mensaje de error debe vincularse a un sistema de ayuda en línea sensible al contexto.

La Figura 16.12 muestra ejemplos de mensajes de error bien y mal diseñados. El mensaje de la izquierda está mal diseñado. Es negativo (acusa al usuario de haber cometido un error), no se adapta a las habilidades y al nivel de experiencia del usuario, y no tiene en cuenta la información del contexto. No sugiere cómo se podría rectificar la situación. Utiliza términos específicos del sistema (identificador de paciente) en vez de un lenguaje orientado al usuario. El mensaje de la derecha es mejor. Es positivo, lo que da a entender que el problema es del sistema y no del usuario. Identifica el problema en términos entendibles para la enfermera y ofrece una forma fácil para corregir el error pulsando un simple botón. El sistema de ayuda está disponible si se necesita.

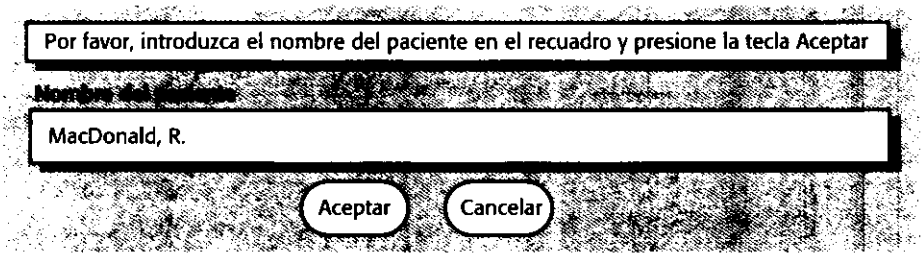


Figura 16.11 Un recuadro de entrada de texto utilizado por una enfermera.

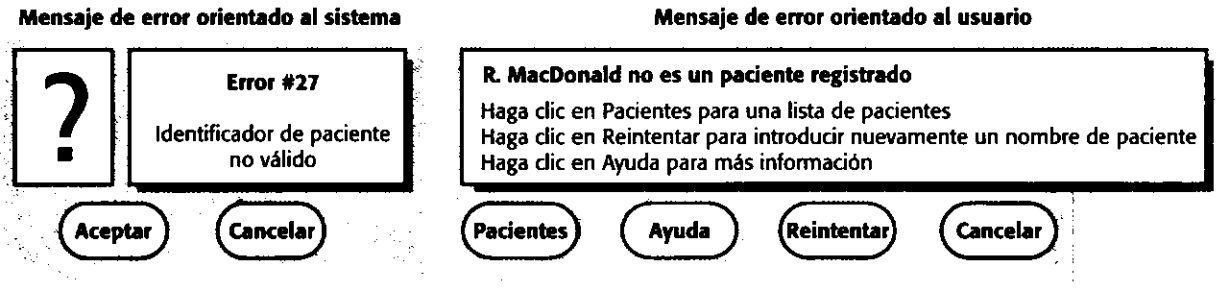


Figura 16.12 Mensajes de error orientados al sistema y al usuario.

16.2 El proceso de diseño de la interfaz de usuario

El diseño de la interfaz de usuario (UI) es un proceso iterativo donde los usuarios interactúan con los diseñadores y prototipos de la interfaz para decidir las características, organización, apariencia y funcionamiento de la interfaz de usuario del sistema. A veces, se construye el prototipo de la interfaz por separado en paralelo con otras actividades de la ingeniería del software. Más comúnmente, en especial cuando se utiliza un desarrollo iterativo, el diseño de la interfaz de usuario se lleva a cabo de forma incremental conforme se desarrolla el software. En ambos casos, sin embargo, antes de que empiece la programación, debe haber desarrollado e, idealmente, probado algunos diseños en papel.

En la Figura 16.13 se ilustra el proceso de diseño general de la UI. Existen tres actividades esenciales en este proceso:

1. *Análisis del usuario.* En el proceso de análisis del usuario, se desarrolla una comprensión de las tareas que éste realiza, su entorno de trabajo, los otros sistemas que utiliza, cómo interactúan con el resto de las personas en su trabajo, etcétera. Para productos con una diversa variedad de usuarios, se debe intentar desarrollar esta comprensión a través de grupos de discusión, pruebas con usuarios potenciales y ejercicios similares.
2. *Prototipado del sistema.* El diseño y desarrollo de la interfaz de usuario es un proceso iterativo. Aunque los usuarios pueden hablar de las facilidades que necesitan de una interfaz, es muy difícil para ellos ser específicos hasta que ven algo tangible. Por lo

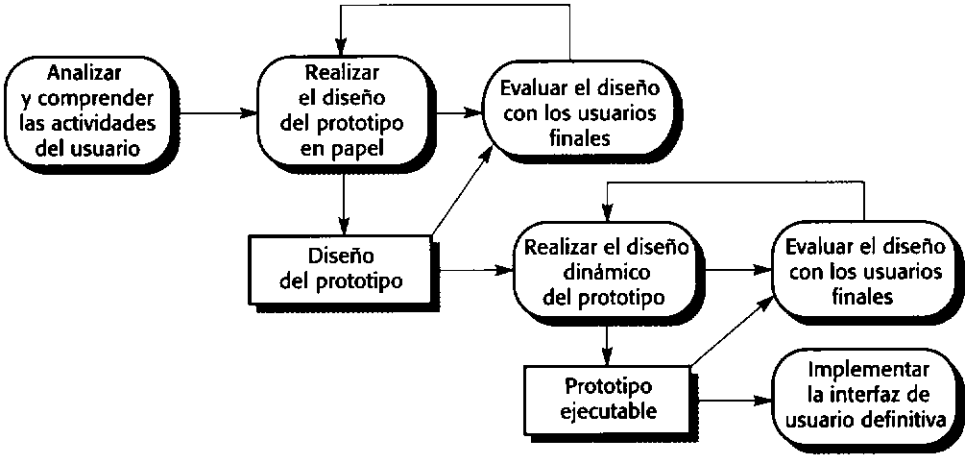


Figura 16.13 El proceso de diseño de la UI.

tanto, se deben desarrollar prototipos del sistema y exponerlos a los usuarios, quienes pueden entonces guiar la evolución de la interfaz.

3. *Evaluación de la interfaz.* Aunque obviamente se habrá hablado con los usuarios durante el proceso de prototipado, también se debe tener una actividad de evaluación más formalizada donde se recopile información sobre las experiencias reales de los usuarios con la interfaz.

Esta sección se centra en el análisis del usuario y en la evaluación de la interfaz, con sólo una breve descripción de las técnicas específicas de prototipado de interfaces de usuario. En el Capítulo 17 se tratan asuntos más generales sobre el prototipado y técnicas de prototipado.

La confección de agendas del diseño de la UI dentro del proceso del software depende, hasta cierto punto, de otras actividades. Como se vio en el Capítulo 7, se puede utilizar la construcción de prototipos como parte del proceso de la ingeniería de requerimientos y, en este caso, tiene sentido empezar el proceso de diseño de la UI en esa etapa. En los procesos iterativos, analizados en el Capítulo 17, el diseño de la UI se integra con el desarrollo del software. Como el software mismo, se puede tener que refactorizar y rediseñar la UI durante el desarrollo.

16.3 Análisis del usuario

Una actividad crítica del diseño de la UI es el análisis de las actividades del usuario que deben ser soportadas por el sistema informático. Si no se entiende lo que los usuarios quieren hacer con el sistema, no se podrá llevar a cabo un diseño real y eficaz de la interfaz de usuario. Para desarrollar esta comprensión, puede utilizar técnicas como el análisis de tareas, estudios etnográficos, entrevistas de usuarios y observaciones o, comúnmente, una mezcla de todas ellas.

Un reto para los ingenieros involucrados en el análisis de usuarios es encontrar una forma de describir los análisis de modo que comuniquen la esencia de las tareas a los otros diseñadores y a los usuarios mismos. Notaciones como los diagramas de secuencia de UML pueden describir las interacciones del usuario y son ideales para comunicarse con los ingenieros de software. Sin embargo, otros usuarios pueden pensar que estos diagramas son demasiado técnicos y no intentarán comprenderlos. Debido a que es muy importante implicar a los usuarios en el proceso de diseño, normalmente habrá que desarrollar escenarios en lenguaje natural para describir las actividades del usuario.

La Figura 16.14 es un ejemplo de un escenario en lenguaje natural que se podría haber desarrollado durante el proceso de especificación y diseño del sistema LIBSYS. Describe una situación en la que no existe el LIBSYS y una estudiante necesita obtener información de otra biblioteca. De este escenario, el diseñador puede ver varios requerimientos:

Jane es una estudiante de estudios religiosos que está preparando una redacción sobre la arquitectura india y cómo ésta ha estado influida por las costumbres religiosas. Para ayudarle a entender esto, le gustaría acceder a imágenes de detalles de edificios notables, pero no puede encontrar nada en su biblioteca local. Se acerca al bibliotecario para exponerle sus necesidades y éste le sugiere términos de búsqueda que podría utilizar. También le sugiere bibliotecas en Nueva Delhi y Londres que podrían tener este material, por lo que entran a los catálogos de la biblioteca y buscan utilizando estos términos. Encuentran alguna fuente de material y hacen una petición de fotocopias de las imágenes con los detalles arquitectónicos, para que se las envíen directamente a Jane.

Figura 16.14 Un escenario de interacción en una biblioteca.

1. Los usuarios podrían no conocer los términos de búsqueda apropiados. Pueden necesitar tener acceso a formas de ayudarles para elegir términos de búsqueda.
2. Los usuarios tienen que poder seleccionar colecciones para buscar.
3. Los usuarios necesitan poder llevar a cabo búsquedas y solicitar copias de material relevante.

No se debe esperar que el análisis del usuario genere requerimientos de interfaces de usuario muy específicos. Normalmente, el análisis ayuda a comprender las necesidades y preocupaciones de los usuarios del sistema. Conforme se conoce mejor cómo trabajan y cuáles son sus preocupaciones y restricciones, éstas pueden tenerse en cuenta en el diseño. Esto significa que es más probable que los diseños iniciales (los cuales, de todos modos, se perfeccionarán a través del prototipado) sean aceptados por los usuarios y, por lo tanto, convencerlos para implicarlos en el proceso de perfeccionamiento del diseño.

16.3.1 Técnicas de análisis

Como se sugirió en la sección anterior, existen tres técnicas base de análisis del usuario: análisis de tareas, entrevistas y cuestionarios, y etnografía. El análisis de tareas y las entrevistas se centran en el individuo y en su trabajo, mientras que la etnografía adopta una perspectiva más general y considera cómo interactúan las personas, cómo organizan su entorno de trabajo y cómo cooperan para resolver problemas.

Existen varias clases de análisis de tareas (Diaper, 1989), pero la más comúnmente utilizada es el Análisis de Tareas Jerárquico (HTA). El HTA fue desarrollado en un principio para ayudar a escribir manuales de usuario, pero también se puede utilizar para identificar lo que hacen los usuarios para alcanzar algún objetivo. En el HTA, una tarea de alto nivel se divide en subtareas, y se identifican planes que especifican lo que pasarían en una situación específica. Empezando con un objetivo del usuario, se dibuja una jerarquía que muestra qué se tiene que hacer para alcanzar ese objetivo. La Figura 16.15 ilustra este enfoque utilizando el escenario de la biblioteca introducido en la Figura 16.14. En la notación del HTA, una línea debajo de la caja normalmente indica que no se descompondrá en subtareas más detalladas.

La ventaja del HTA sobre los escenarios en lenguaje natural es que obliga a considerar cada una de las tareas y decidir si éstas se deben descomponer. Con los escenarios en lenguaje natural, es fácil omitir tareas importantes. Los escenarios también se hacen largos y pesados de leer si se desea añadirles muchos detalles.

El problema con este enfoque para describir las tareas del usuario es que es más apropiado para tareas que son procesos secuenciales. La notación se hace difícil cuando se intenta modelar tareas que implican actividades entrelazadas o simultáneas o que conllevan muchas subtareas. Además, el HTA no registra por qué las tareas se hacen de una forma en particular o restringen los procesos del usuario. Se puede obtener una visión parcial de las actividades del usuario a través del HTA, pero se necesita información adicional para desarrollar una comprensión más completa de los requerimientos de diseño de la UI.

Normalmente, se recopila información para el HTA a través de la observación y de entrevistas con los usuarios. En este proceso de entrevistas, se puede recopilar parte de esta información adicional y registrarla al lado de los análisis de tareas. Cuando se realizan entrevistas para descubrir lo que los usuarios hacen realmente, se deben diseñar las entrevistas de forma que los usuarios puedan proporcionar información que ellos (en vez del entrevistador) piensen que es importante. Esto significa que no hay que ceñirse rígidamente a una lista preparada de preguntas. Más bien, las preguntas deben ser abiertas y animar a los usuarios a que digan por qué hacen las cosas además de lo que realmente hacen.

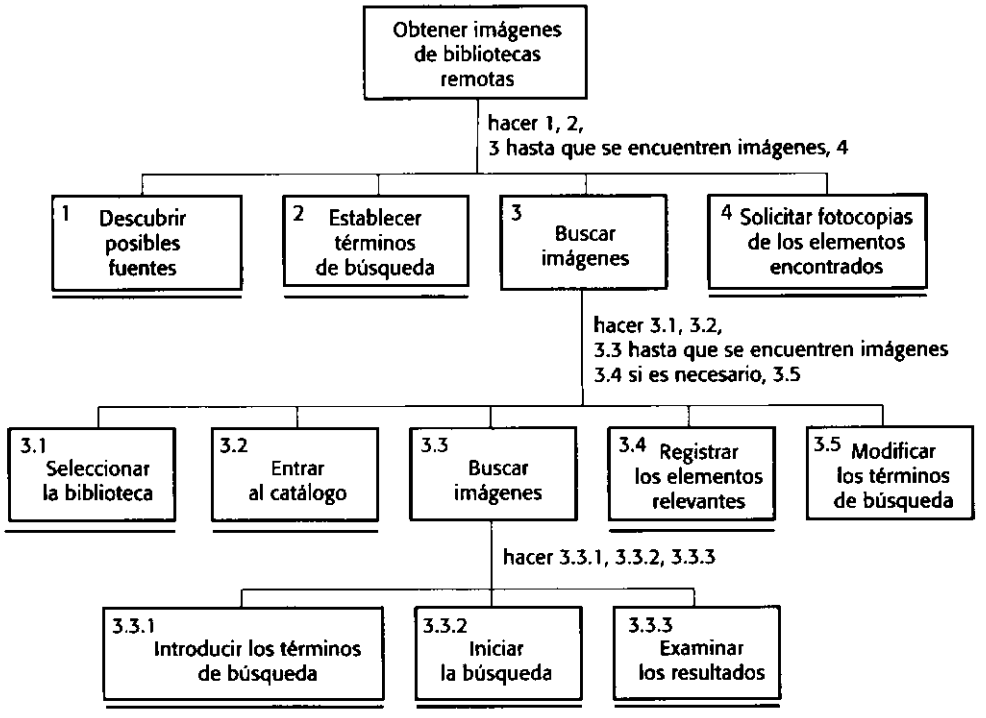


Figura 16.15
Análisis de tareas jerárquico.

Las entrevistas, por supuesto, no son sólo una forma de obtener información para el análisis de tareas: son una técnica general para obtener información. Se pueden complementar las entrevistas individuales con entrevistas en grupo o grupos de discusión. La ventaja de utilizar grupos de discusión es que los usuarios se estimulan entre sí para proporcionar información y pueden terminar discutiendo diferentes formas que han desarrollado de utilizar los sistemas.

El análisis de tareas se centra en cómo trabajan los individuos, pero, por supuesto, la mayoría del trabajo es realmente cooperativo. Las personas trabajan juntas para conseguir un objetivo, y los usuarios encuentran difícil discutir cómo se lleva a cabo esta colaboración. Por lo tanto, la observación directa de cómo trabajan y utilizan los sistemas informáticos los usuarios es una importante técnica adicional de análisis del usuario.

Un enfoque para la observación directa que ha sido utilizado en una amplia variedad de escenarios es la etnografía (Suchman, 1983; Hughes *et al.*, 1997; Crabtree, 2003). La etnografía se trató en el Capítulo 7 como una técnica que apoya a la ingeniería de requerimientos. Los etnógrafos observan de cerca cómo trabajan las personas, cómo se relacionan entre sí y cómo utilizan los recursos de su lugar de trabajo para ayudarlas en él. La ventaja de la etnografía es que los etnógrafos pueden observar las acciones intuitivas y las colaboraciones informales, que pueden entonces provocar nuevas discusiones sobre el trabajo.

Como ejemplo de cómo puede influir la etnografía en el diseño de las interfaces de usuario, la Figura 16.16 es un fragmento de un informe de un estudio etnográfico de los controladores del tráfico aéreo en el cual estuve implicado (Bentley *et al.*, 1992). Estábamos interesados en el diseño de la interfaz para un sistema de control del tráfico aéreo (ATC) más automatizado y aprendimos dos cosas importantes de estas observaciones:

1. Los controladores no podían ver todos los vuelos de un sector (esta era la razón por la que extendían tiras en la mesa). Por lo tanto, debemos evitar utilizar vistas que em-

El control del tráfico aéreo implica varios «juegos» de controles donde los juegos que controlan sectores adyacentes del espacio aéreo están físicamente situados unos al lado de otros. Los vuelos en un sector se representan mediante tiras de papel que se ajustan en rejillas de madera en un orden que refleja su posición en el sector. Si no hay suficientes ranuras en la rejilla (por ejemplo, cuando hay mucho tráfico en el espacio aéreo), los controladores extienden las tiras en la mesa delante de la rejilla. Cuando observábamos a los controladores, nos dimos cuenta de que con regularidad miraban las rejillas de tiras del sector adyacente. Se lo comentamos y les preguntamos por qué lo hacían. Respondieron que, cuando el controlador adyacente tiene tiras en su mesa, significa que están entrando muchos vuelos en sus sectores. Por lo tanto, intentaban incrementar la velocidad de los aviones en el sector para «despejar el espacio» para los aviones entrantes.

Figura 16.16
Un informe de observaciones del control del tráfico aéreo.

pleen barras de desplazamiento donde los vuelos desaparezcan por la parte superior o inferior de la vista.

2. La interfaz debe tener alguna forma de comunicar a los controladores cuántos vuelos hay en los sectores adyacentes de forma que puedan planificar su carga de trabajo.

Comprobar los sectores adyacentes era una acción automática del controlador y es muy probable que no la hubieran mencionado en las discusiones sobre el proceso del ATC. Descubrimos estos importantes requerimientos sólo a través de la observación directa.

Ninguna de estas técnicas de análisis del usuario, por sí misma, proporciona una visión completa de lo que realmente hacen los usuarios. Éstas son enfoques complementarios que deben utilizarse para ayudar a entender lo que hacen los usuarios y a comprender mejor lo que podría ser un diseño de la interfaz de usuario apropiado.

16.4 Prototipado de la interfaz de usuario

Debido a la naturaleza dinámica de las interfaces de usuario, las descripciones textuales y los diagramas no son adecuados para expresar los requerimientos de éstas. El prototipado evolutivo o exploratorio con la implicación de los usuarios finales es la única forma práctica de diseñar y desarrollar interfaces gráficas de usuario para sistemas software. Implicar al usuario en el proceso de diseño y desarrollo es un aspecto fundamental del *diseño centrado en el usuario* (Normal y Draper, 1986), un criterio de diseño para sistemas interactivos.

El propósito del prototipado es permitir a los usuarios adquirir una experiencia directa con la interfaz. La mayoría de nosotros encuentra difícil pensar de forma abstracta sobre una interfaz de usuario y explicar exactamente qué deseamos. Sin embargo, cuando se nos presentan ejemplos, es fácil identificar las características que nos gustan y las que no.

Idealmente, cuando se está construyendo el prototipo de una interfaz de usuario, se debe adoptar un proceso de prototipado en dos etapas:

1. Al principio del proceso, hay que desarrollar prototipos en papel —maquetas de los diseños de las pantallas— y mostrárselos a los usuarios finales.
2. Entonces, se perfecciona el diseño y se desarrollan prototipos automatizados cada vez más sofisticados, y se ponen a disposición de los usuarios para realizar pruebas y simulación de actividades.

La construcción de prototipos en papel es un enfoque poco costoso y sorprendentemente efectivo para el desarrollo de prototipos (Snyder, 2003). No se necesita desarrollar ningún software ejecutable y los diseños no tienen por qué hacerse conforme a estándares profesionales. Se pueden hacer versiones en papel de las pantallas del sistema con las que interactúan los usuarios y se puede diseñar un conjunto de escenarios que describan cómo se podría utilizar el sistema. Conforme se desarrolla un escenario, hay que esbozar la información que se mostrará y las opciones disponibles a los usuarios.

Se debe trabajar entonces a través de estos escenarios con los usuarios para simular la manera en que se utilizará el sistema. Es una forma efectiva de ver las reacciones iniciales de los usuarios a un diseño de la interfaz, la información que necesitan del sistema y cómo interactuarán normalmente con el sistema.

Como alternativa, se puede utilizar una técnica basada en *storyboards* para presentar el diseño de la interfaz. Un *storyboard* es una serie de esbozos que ilustran una secuencia de interacciones. Esto es menos manejable, pero puede ser de mayor utilidad cuando se presentan las propuestas de interfaz a grupos en vez de a personas individuales.

Después de los experimentos iniciales con los prototipos en papel, se debe implementar un prototipo software del diseño de la interfaz. El problema, por supuesto, es que se necesita que el sistema tenga alguna funcionalidad con la cual los usuarios puedan interactuar. Si se está construyendo el prototipo de la UI al principio del proceso de desarrollo del sistema, puede ser que esta funcionalidad no esté disponible. Para evitar este problema, se puede utilizar el prototipado de «Mago de Oz» (si no se ha visto la película, véase la página web para una explicación). En este enfoque, los usuarios interactúan con lo que parece ser un sistema informático, pero sus entradas se canalizan a una persona oculta que simula las respuestas del sistema. Pueden hacer esto directamente o utilizando algún otro sistema para calcular las respuestas requeridas. En este caso, no se necesita tener ningún software ejecutable aparte de la interfaz de usuario propuesta.

Se pueden llevar a cabo experimentos de prototipado adicionales utilizando tanto un enfoque evolutivo como un enfoque desechable. Estos enfoques de prototipado analizan en el Capítulo 17, donde también se describen varias técnicas que se pueden emplear para el prototipado y el desarrollo rápido de aplicaciones. Existen tres enfoques que pueden utilizarse para el prototipado de interfaces de usuario:

1. *Enfoque dirigido por secuencias de comandos.* Si solamente se necesita estudiar ideas con los usuarios, se puede utilizar un enfoque dirigido por secuencias de comandos, como el que encontrará en Macromedia Director. En este enfoque, se crean pantallas con elementos visuales, como botones y menús, y se asocia una secuencia de comandos con estos elementos. Cuando el usuario interactúa con estas pantallas, se ejecuta la secuencia de comandos y se presenta la siguiente pantalla, que les muestra los resultados de sus acciones. No hay implicada ninguna aplicación lógica.
2. *Lenguajes de programación visuales.* Los lenguajes de programación visuales, como Visual Basic, incorporan un potente entorno de desarrollo, acceden a una gran variedad de objetos reutilizables y a un sistema de desarrollo de interfaces de usuario que permite crear interfaces de forma rápida, con componentes y secuencias de comandos asociados con los objetos de la interfaz. Se describen los sistemas de desarrollo visuales en el Capítulo 17.
3. *Prototipado basado en Internet.* Estas soluciones, basadas en navegadores web y en lenguajes como Java, ofrecen una interfaz de usuario hecha. Se añade funcionalidad asociando segmentos de programas Java con la información a visualizar. Estos segmentos (llamados applets) se ejecutan automáticamente cuando se carga la página en

el navegador. Este enfoque es una forma rápida de desarrollar prototipos de interfaces de usuario, pero existen restricciones inherentes impuestas por el navegador y por el modelo de seguridad de Java.

Obviamente, el prototipado está muy relacionado con la evaluación de la interfaz. Es improbable que las evaluaciones formales sean económicas para los primeros prototipos, ya que lo que se está intentando conseguir en esta etapa es una «evaluación formativa» en la que se buscan formas de mejorar la interfaz. Conforme el prototipo se hace más completo, pueden utilizar técnicas de evaluación sistemática, como se verá en la siguiente sección.

16.5 Evaluación de la interfaz

La evaluación de la interfaz es el proceso de evaluar la forma en que se utiliza una interfaz y verificar que cumple los requerimientos del usuario. Por lo tanto, debe ser parte del proceso de verificación y validación de los sistemas software. Neilsen (Neilsen, 1993), en su libro sobre ingeniería de usabilidad, incluye un buen capítulo sobre este tema.

De forma ideal, una evaluación se debe llevar a cabo contra una especificación de la usabilidad basada en atributos de la usabilidad, como se muestra en la Figura 16.17. Las métricas de estos atributos de usabilidad se pueden idear. Por ejemplo, en una métrica de aprendizaje, se podría enunciar que a un operador familiarizado con las tareas implementadas le debe ser posible utilizar el 80% de la funcionalidad del sistema después de tres horas de formación. Sin embargo, es más común especificar la usabilidad (si es que se especifica del todo) de forma cuantitativa en vez de utilizar métricas. Por lo tanto, el diseñador tiene que utilizar su criterio y experiencia en la evaluación de las interfaces.

La evaluación sistemática del diseño de la interfaz de usuario puede ser un proceso caro que implica a científicos cognoscitivos y diseñadores gráficos. Es posible que se tenga que diseñar y realizar un número estadísticamente importante de experimentos con los usuarios típicos. Se puede necesitar el uso de laboratorios construidos especialmente con equipos de supervisión. Una evaluación de la interfaz de usuario de este tipo es económicamente poco realista para sistemas desarrollados por pequeñas organizaciones con recursos limitados.

Existen varias técnicas menos costosas y sencillas en la evaluación de interfaces que pueden identificar deficiencias específicas en el diseño de interfaces:

1. Cuestionarios que recopilan información de lo que opinan los usuarios de la interfaz.
2. La observación de los usuarios cuando trabajan con el sistema y «piensan en voz alta» de cómo tratan de utilizar el sistema para llevar a cabo alguna tarea.

Atributo	Descripción
Aprendizaje	¿Cuánto tiempo tarda un usuario nuevo en ser productivo con el sistema?
Velocidad de funcionamiento	¿Cómo responde el sistema a las operaciones de trabajo del usuario?
Robustez	¿Qué tolerancia tiene el sistema a los errores del usuario?
Recuperación	¿Cómo se recupera el sistema a los errores del usuario?
Adaptación	¿Está muy atado el sistema a un único modelo de trabajo?

Figura 16.17 Atributos de usabilidad.

3. «Instantáneas» de vídeos del uso típico del sistema.
4. La inclusión de código en el software que recopila información de los recursos más utilizados y de los errores más comunes.

Examinar a los usuarios utilizando un cuestionario es una forma relativamente económica de evaluar una interfaz. Las preguntas deben ser precisas más que generales. No es de utilidad hacer preguntas como «Por favor, haga comentarios sobre la usabilidad de la interfaz», puesto que las respuestas probablemente variarán tanto que no se descubrirá ninguna tendencia común. En su lugar, las preguntas específicas como «Por favor, indique el valor en una escala de 1 a 5 de cuál es la comprensión de los mensajes de error. Un valor 1 significa muy clara y 5 significa incomprensible» son mejores. Son más fáciles de responder y es más probable que proporcionen información útil para mejorar la interfaz.

Cuando están rellenando el cuestionario, a los usuarios se les debe preguntar sobre su experiencia y conocimientos. Esto permite al diseñador saber si los usuarios con cierto tipo de conocimientos tienen problemas con la interfaz. Los cuestionarios se pueden utilizar aun antes de que esté disponible el sistema ejecutable si se construyen y evalúan maquetas en papel de la interfaz.

La evaluación basada en la observación sencillamente comprende observar cómo utilizan el sistema los usuarios, ver los recursos que utilizan, los errores cometidos, etcétera. Esto se puede complementar con sesiones de «pensar en voz alta», en las cuales los usuarios conversan sobre lo que tratan de hacer, qué piensan del sistema y cómo tratan de utilizarlo para llevar a cabo sus objetivos.

Utilizar equipo de vídeo de relativamente bajo coste significa que puede grabar las sesiones de usuario para su análisis posterior. Un análisis completo por medio de vídeo es caro y requiere un equipo de evaluación especializado con varias cámaras enfocadas al usuario y a la pantalla. Sin embargo, la grabación en vídeo de algunas operaciones específicas puede ayudar a detectar los problemas. Se deben utilizar otros métodos de evaluación para detectar qué operaciones provocan dificultades al usuario.

El análisis de las grabaciones permite al diseñador descubrir si la interfaz requiere demasiado movimiento de las manos (un problema con algunos sistemas es que los usuarios deben mover frecuentemente sus manos del teclado al ratón) y ver si son necesarios movimientos forzados del ojo. Una interfaz que requiera muchos cambios de enfoque puede implicar que los usuarios cometan más errores y pierdan partes de la visualización.

Instrumentar código que recopile estadísticas de utilización permite mejorar las interfaces de varias formas. Se pueden detectar las operaciones más comunes. Las interfaces se pueden reorganizar para que sean más rápidas de seleccionar. Por ejemplo, si se utilizan menús contextuales o descendentes, las operaciones más frecuentes se deben ubicar en la parte superior del menú y las operaciones destructivas en la parte inferior. La instrumentación de código también permite que los comandos propensos a errores se detecten y modifiquen.

Finalmente, es fácil proporcionar a los usuarios un comando que puedan utilizar para enviar mensajes al diseñador de la herramienta. Esto hace que los usuarios sientan que sus opiniones son tenidas en cuenta. Así, el diseñador de la interfaz y otros ingenieros pueden obtener una rápida retroalimentación de los problemas particulares.

Ninguno de estos enfoques relativamente simples para la evaluación de la interfaz de usuario es infalible y probablemente no detectan todos los problemas de las interfaces de usuario. Sin embargo, las técnicas se pueden utilizar con un grupo de voluntarios sin un gran desembolso de recursos antes de que se entregue el sistema. Así, se pueden descubrir y corregir muchos de los peores problemas del diseño de las interfaces de usuario.

PUNTOS CLAVE

- Los principios de las interfaces de usuario que abarcan la familiaridad del usuario, la uniformidad, la mínima sorpresa, la recuperabilidad, la guía al usuario y la diversidad de usuarios ayudan a guiar el diseño de las interfaces de usuario.
- Los estilos de interacción con un sistema software incluyen la manipulación directa, los sistemas de menús, el rellenado de formularios, los lenguajes de comandos y el lenguaje natural.
- La visualización gráfica de la información se debe utilizar cuando se pretenda presentar tendencias y valores aproximados. La visualización digital sólo se debe utilizar cuando se requiere precisión.
- El color se debe utilizar con moderación y de forma uniforme en las interfaces de usuario. Los diseñadores deben tener en cuenta el hecho de que un importante número de personas son daltónicas.
- El proceso de diseño de la interfaz de usuario incluye subprocesos relacionados con el análisis, el prototipado de la interfaz y la evaluación de ésta.
- El objetivo del análisis del usuario es dar a conocer a los diseñadores las formas de trabajar reales de los usuarios. Debe utilizar diferentes técnicas —análisis de tareas, entrevistas y observación— durante el análisis del usuario.
- El desarrollo de prototipos de interfaces de usuario debe ser un proceso en etapas con prototipos iniciales basados en versiones en papel de la interfaz que, después de una evaluación y retroalimentación inicial, se utilizan como base para prototipos automatizados.
- Los objetivos de la evaluación de interfaces de usuario son obtener una retroalimentación de cómo se puede mejorar el diseño de la UI y evaluar si una interfaz cumple sus requerimientos de usabilidad.

LECTURAS ADICIONALES

Human-Computer Interaction, 3rd ed. Un buen texto general cuyo punto fuerte se centra en los temas de diseño y el trabajo cooperativo. (A. Dix *et al.*, 2004, Prentice-Hall.)

Interaction Design. Este libro se centra en el diseño de la interacción con los sistemas informáticos. Presenta gran parte del mismo material que *Human-Computer Interaction* pero de una forma bastante diferente. Ambos están bien escritos y merece la pena leerlos. (J. Preece *et al.*, 2002, John Wiley & Sons.)

«Usability Engineering». Este número especial de *IEEE Software* incluye varios artículos sobre usabilidad escritos especialmente para lectores con conocimientos sobre ingeniería del software. [*IEEE Software*, 18(1), enero 2001.]

EJERCICIOS

16.1 En la Sección 16.1 se sugirió que los objetos que manipulan los usuarios se deben obtener de su dominio y no del dominio de la informática. Sugiera objetos adecuados para los siguientes usuarios y sistemas:

- Un ayudante de almacén que utiliza un catálogo de piezas automatizado.
- Un piloto de aviones que utiliza un sistema de supervisión de seguridad de aviones.

- Un gerente que manipula una base de datos financiera.
- Un policía que utiliza un sistema de control de coches patrulla.

16.2 Sugiera situaciones donde no sea adecuado o posible proveer una interfaz de usuario uniforme.

16.3 ¿Qué factores deben tenerse en cuenta cuando se diseña una interfaz basada en menús para sistemas del tipo de un cajero automático (ATM) de un banco? Redacte un comentario crítico sobre la interfaz de un ATM que utilice.

16.4 Sugiera formas en las que podría adaptar la interfaz de usuario de un sistema de comercio electrónico, como una librería en línea o una tienda de discos, a usuarios que tienen problemas visuales o de control muscular.

16.5 Señale las ventajas de la visualización gráfica de la información y sugiera cuatro aplicaciones donde sea más apropiado utilizar vistas gráficas en vez de vistas digitales de información numérica.

16.6 ¿Cuáles son las pautas a seguir cuando se utiliza color en una interfaz de usuario? Sugiera cómo utilizar el color de forma más efectiva en la interfaz de alguna aplicación empleada frecuentemente.

16.7 Considere los mensajes de error producidos por MS-Windows, Linux, Mac Os o algún otro sistema operativo. Sugiera cómo mejorar estos mensajes.

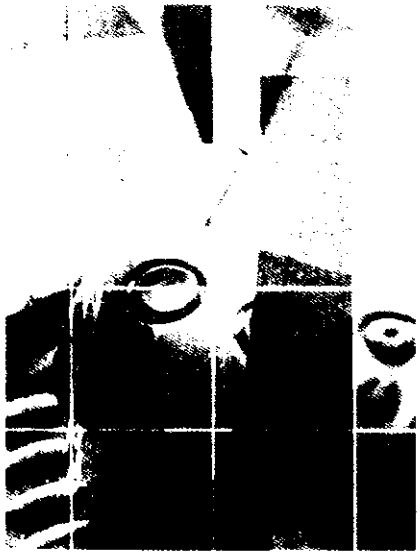
16.8 Escriba posibles escenarios de interacción para los siguientes sistemas:
- La utilización de un servicio basado en web de reserva de entradas para el teatro con el fin de solicitar entradas y pagarlas mediante tarjeta de crédito.
- La solicitud de las mismas entradas utilizando una interfaz sobre teléfono móvil.
- La utilización de un conjunto de herramientas CASE para crear un modelo de objetos de un sistema software (véanse los Capítulos 8 y 14) y la generación automática de código del modelo.

16.9 ¿En qué circunstancias podría utilizar el prototipado de «Mago de Oz»? ¿Para qué tipo de sistemas es inapropiado este enfoque?

16.10 Diseñe un cuestionario que le permita recoger información sobre la interfaz de usuario de alguna herramienta (como un procesador de textos) con la que esté familiarizado. Si es posible, distribuya este cuestionario a varios usuarios y trate de evaluar los resultados. ¿Qué dicen éstos sobre el diseño de la interfaz de usuario?

16.11 Comente si es ético implementar software para controlar su utilización sin decirles a los usuarios que se está controlando su trabajo.

16.12 ¿Cuáles son los puntos éticos a los que podrían enfrentarse los diseñadores de interfaces cuando tratan de compaginar las necesidades de los usuarios finales de un sistema con las necesidades de la organización que está pagando por el sistema a desarrollar?



PARTE

I DESARROLLO

Cuando por primera vez se estableció la ingeniería del software como una disciplina, el proceso de desarrollo de la mayoría de los sistemas era un proceso de escribir un programa basado en una especificación del diseño. Se utilizaban lenguajes de programación imperativos como C, FORTRAN o Ada. En los textos de ingeniería del software, los capítulos sobre el desarrollo de software se centraban principalmente en las buenas prácticas de programación.

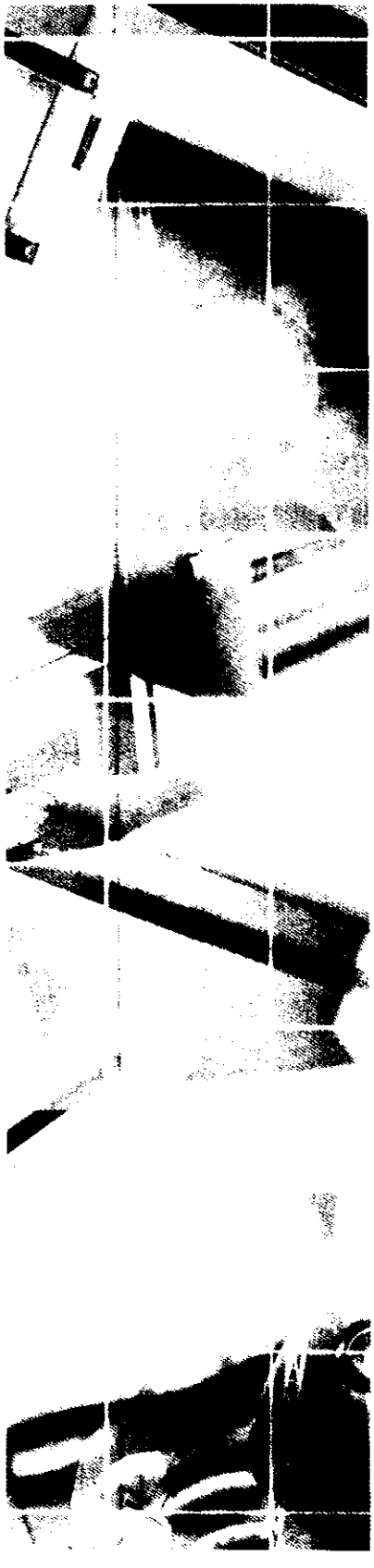
En la actualidad existen diferentes formas de desarrollar software. Entre ellas se encuentran la programación original en lenguajes como C++ o Java, la generación de secuencias de comandos, la programación de bases de datos, la generación de programas a través de herramientas CASE, y la ingeniería del software basada en la reutilización. Además, se está reconociendo finalmente el hecho de que existe una distinción real entre el desarrollo y el mantenimiento, y estamos empezando a pensar en el desarrollo como la primera etapa en un proceso de la evolución del programa. Para reflejar estos desarrollos, se ha incluido esta parte nueva en el libro, centrada en las técnicas de desarrollo. Existen cinco capítulos en esta parte:

1. El Capítulo 17 es un capítulo nuevo que describe las técnicas para el desarrollo rápido de software. En el entorno de negocios de hoy día significa que las compañías necesitan que su software se entregue rápidamente para que puedan responder a nuevos desafíos y oportunidades. En este capítulo, se analizan los métodos ágiles de desarrollo, centrando la atención de modo especial en la programación extrema. También se describen entornos para el desarrollo rápido de aplicaciones y la adecuada utilización del prototipado de sistemas.

2. Los Capítulos 18 y 19 tratan de la ingeniería del software basada en la reutilización. Durante los últimos años, la reutilización del software se ha hecho cada vez más común y el desarrollo basado en la reutilización es actualmente un enfoque dominante en la ingeniería del software. El Capítulo 18 presenta una visión general de la reutilización del software y el desarrollo con reutilización. El Capítulo 19 se centra en la ingeniería del software basada en componentes, incluyendo la composición de componentes y el proceso CBSE.

3. El Capítulo 20 continúa la exposición de los sistemas críticos que está presente en todo el libro. Se tratan varios enfoques de desarrollo para conseguir la confiabilidad del sistema, incluyendo la evitación de defectos y la tolerancia a los mismos, y se muestra cómo las construcciones y técnicas de programación se pueden utilizar para conseguir la confiabilidad. En la parte final de este capítulo, se vuelve al tema de la arquitectura del software y se describen los enfoques arquitectónicos para la tolerancia a defectos.

4. El Capítulo 21 trata sobre la evolución del software. Los cambios son inevitables en todos los sistemas software y, en vez de considerar el proceso de cambio como una actividad separada, pienso que tiene sentido considerarlo como una continuación del desarrollo inicial del software. En este capítulo, se estudia la inevitabilidad de la evolución, el mantenimiento del software, los procesos de la evolución y la toma de decisiones para la evolución de los sistemas heredados.



17 Desarrollo rápido de software

Objetivos

El objetivo de este capítulo es describir varios enfoques para el desarrollo de software pensados para la entrega rápida del software. Cuando haya leído este capítulo:

- entenderá cómo un enfoque de desarrollo de software iterativo e incremental conduce a una entrega más rápida de un software más útil;
- entenderá las diferencias entre los métodos de desarrollo ágiles y los métodos de desarrollo de software que dependen de la documentación de las especificaciones y diseños;
- conocerá los principios, prácticas y algunas de las limitaciones de la programación extrema;
- entenderá cómo se puede utilizar el prototipado para ayudar a resolver requerimientos y diseñar incertidumbres cuando se tiene que utilizar un enfoque de desarrollo basado en la especificación.

Contenidos

Actualmente, los negocios operan en un entorno global que cambia rápidamente. Tienen que responder a nuevas oportunidades y mercados, condiciones económicas cambiantes y la aparición de productos y servicios competidores. El software es parte de casi todas las operaciones de negocio, por lo que es fundamental que el software nuevo se desarrolle rápidamente para aprovechar nuevas oportunidades y responder a la presión competitiva. Por lo tanto, actualmente el desarrollo y entrega rápidos son a menudo los requerimientos más críticos de los sistemas software. De hecho, muchas compañías están dispuestas a una pérdida en la calidad del software y en el compromiso sobre los requerimientos en favor de una entrega rápida del software.

Debido a que estas compañías operan en un entorno cambiante, a menudo es prácticamente imposible obtener un conjunto completo de requerimientos del software estables. Los requerimientos que se proponen cambian inevitablemente, porque a los clientes les resulta imposible predecir cómo afectará un sistema a la manera de trabajar, cómo interactuará con otros sistemas y qué operaciones de los usuarios se deben automatizar. Es posible que los requerimientos reales sólo queden claros cuando se haya entregado el sistema y los usuarios hayan adquirido experiencia.

Los procesos de desarrollo del software basados en una completa especificación de los requerimientos y posterior diseño, construcción y pruebas del sistema no se ajustan al desarrollo rápido de aplicaciones. Cuando los requerimientos cambian o cuando se descubren problemas con ellos, el diseño o implementación del sistema se tiene que volver a realizar o probar. Como consecuencia, normalmente se prolonga en el tiempo un proceso en cascada convencional o basado en la especificación y el software definitivo se entrega al cliente mucho tiempo después de que fuera inicialmente especificado.

En un entorno de negocios que se mueve con rapidez, esto puede causar verdaderos problemas. Para cuando esté disponible el software, la razón original de su adquisición puede haber cambiado tan radicalmente que el software sea en realidad inútil. Por lo tanto, en particular para los sistemas de negocio, los procesos de desarrollo que se basan en el desarrollo y entrega rápidos de software son esenciales.

Los procesos de desarrollo rápido de software están diseñados para producir software útil de forma rápida. Generalmente, son procesos iterativos en los que se entrelazan la especificación, el diseño, el desarrollo y las pruebas. El software no se desarrolla y utiliza en su totalidad, sino en una serie de incrementos, donde en cada incremento se incluyen nuevas funcionalidades al sistema. Aunque existen muchos enfoques para el desarrollo rápido de software, comparten las mismas características fundamentales:

1. Los procesos de especificación, diseño e implementación son concurrentes. No existe una especificación del sistema detallada, y la documentación del diseño se minimiza o es generada automáticamente por el entorno de programación utilizado para implementar el sistema. El documento de requerimientos del usuario define solamente las características más importantes del sistema.
2. El sistema se desarrolla en una serie de incrementos. Los usuarios finales y otros stakeholders del sistema participan en la especificación y evaluación de cada incremento. Pueden proponer cambios en el software y nuevos requerimientos que se deben implementar en un incremento posterior del sistema.
3. A menudo se desarrollan las interfaces de usuario del sistema utilizando un sistema de desarrollo interactivo que permite que el diseño de la interfaz se cree rápidamente dibujando y colando iconos en la interfaz. El sistema puede generar una interfaz basada en web para un navegador o una interfaz para una plataforma específica como Microsoft Windows.

El desarrollo incremental, introducido en el Capítulo 4, implica producir y entregar el software en incrementos más que en un paquete único. Cada iteración del proceso produce un nuevo incremento del software. Las dos ventajas principales de adoptar un enfoque incremental para el desarrollo del software son:

1. *Entrega acelerada de los servicios del cliente.* En los incrementos iniciales del sistema se pueden entregar las funcionalidades de alta prioridad para que los clientes puedan aprovechar el sistema desde el principio de su desarrollo. Los clientes pueden ver sus requerimientos en la práctica y especificar cambios a incorporar en entregas posteriores del sistema.
2. *Compromiso del cliente con el sistema.* Los usuarios del sistema tienen que estar implicados en el proceso de desarrollo incremental debido a que tienen que proporcionar retroalimentación sobre los incrementos entregados al equipo de desarrollo. Su participación no sólo significa que es más probable que el sistema cumpla sus requerimientos, sino que también los usuarios finales del sistema tienen que hacer un compromiso con él y conseguir que éste llegue a funcionar.

En la Figura 17.1 se ilustra un modelo de proceso general para el desarrollo incremental. Observe que las etapas iniciales de este proceso se centran en el diseño arquitectónico. Si no se considera la arquitectura al principio del proceso, es probable que la estructura general del sistema sea inestable y se degrade conforme se entreguen nuevos incrementos.

El desarrollo incremental del software es un enfoque mucho mejor para el desarrollo de la mayoría de los sistemas de negocio, comercio electrónico y personales porque refleja el modo fundamental al que todos nosotros tendemos al resolver problemas. Rara vez encontramos una solución completa a un problema por adelantado, pero nos movemos hacia una solución en una serie de pasos, dando marcha atrás cuando nos damos cuenta de que hemos cometido un error.

Sin embargo, puede haber verdaderos problemas con este enfoque, particularmente en las grandes compañías con procedimientos bastante rígidos y en organizaciones donde el desarrollo del software normalmente se subcontrata con un contratista exterior. Los principales problemas con el desarrollo iterativo y la entrega incremental son:

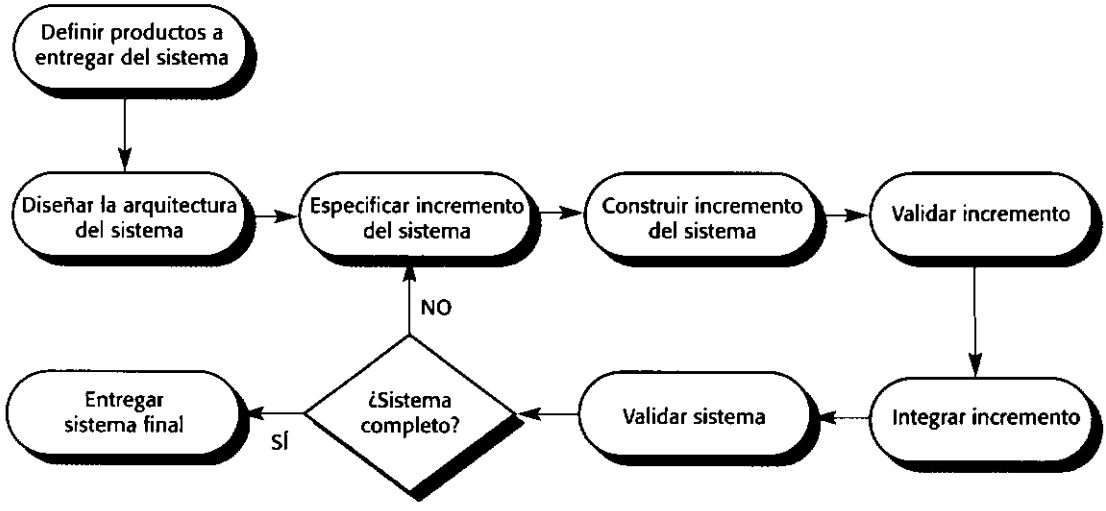


Figura 17.1 Un proceso de desarrollo iterativo.

1. *Problemas de administración.* Las estructuras de administración del software para sistemas grandes se diseñan para tratar con modelos de proceso del software que generan entregas periódicas para evaluar el progreso. Los sistemas desarrollados incrementalmente cambian tan rápido que no es rentable producir una gran cantidad de documentación del sistema. Además, el desarrollo incremental muchas veces puede requerir el uso de tecnologías desconocidas para asegurar una entrega más rápida del software. A los administradores puede resultarles difícil utilizar al personal existente en los procesos de desarrollo incremental puesto que carecen de las habilidades requeridas.
2. *Problemas contractuales.* El modelo contractual normal entre un cliente y un desarrollador de software se basa en la especificación del sistema. Cuando no existe tal especificación, puede ser difícil diseñar un contrato para el desarrollo del sistema. Los clientes pueden estar descontentos con un contrato que simplemente pague a los desarrolladores por el tiempo invertido en el proyecto, ya que puede conducir a que el sistema se desarrolle lentamente y se sobrepase el presupuesto; los desarrolladores probablemente no aceptarán un contrato con precio fijo debido a que no pueden controlar los cambios requeridos por los usuarios finales.
3. *Problemas de validación.* En un proceso basado en la especificación, la verificación y la validación están pensadas para demostrar que el sistema cumple su especificación. Un equipo independiente de V & V puede empezar a trabajar tan pronto como esté disponible la especificación y puede preparar pruebas en paralelo con la implementación del sistema. Los procesos de desarrollo iterativo intentan minimizar la documentación y entrelazan la especificación y el desarrollo. Por lo tanto, la validación independiente de los sistemas desarrollados incrementalmente es difícil.
4. *Problemas de mantenimiento.* Los cambios continuos tienden a corromper la estructura del cualquier sistema software. Esto significa que cualquiera, aparte de los desarrolladores originales, puede tener dificultades para entender el software. Una forma de reducir este problema es utilizar refactorización, donde se mejoran continuamente las estructuras del software durante el proceso de desarrollo. Esto se expone en la Sección 17.2, donde se trata la programación extrema. Además, si se utiliza tecnología especializada, como los entornos RAD (descritos en la Sección 17.3), para ayudar al desarrollo rápido del prototipo, la tecnología RAD puede convertirse en obsoleta. Por lo tanto, puede ser difícil encontrar personas que tengan los conocimientos requeridos para dar mantenimiento al sistema.

Por supuesto, existen algunos tipos de sistemas donde el desarrollo y entrega rápidos no son el mejor enfoque. Éstos son sistemas muy grandes donde el desarrollo puede implicar equipos que trabajan en diferentes lugares, algunos sistemas embebidos donde el software depende del desarrollo del hardware y algunos sistemas críticos en los que se deben analizar todos los requerimientos para verificar las interacciones que puedan comprometer la seguridad o protección del sistema.

Estos sistemas, por supuesto, sufren los mismos problemas de requerimientos inciertos y cambiantes. Por lo tanto, para abordar estos problemas y conseguir algunos de los beneficios del desarrollo incremental, se puede utilizar un proceso híbrido en el que se desarrolle de forma iterativa un prototipo del sistema y se utilice como una plataforma para experimentar con los requerimientos y diseño del sistema. Con la experiencia adquirida con el prototipo, se puede tener una mayor seguridad de que los requerimientos cumplen las necesidades reales de los stakeholders del sistema.

Se utiliza aquí el término *prototipado* para expresar un proceso iterativo para desarrollar un sistema experimental que *no* está destinado a la utilización por parte de los clientes. Se desarrolla un prototipo del sistema para ayudar a los desarrolladores de software y a los clientes a comprender qué se debe implementar. Sin embargo, algunas veces se utiliza la expresión *prototipado evolutivo* como sinónimo del desarrollo de software incremental. El prototipo no se descarta sino que se desarrolla para cumplir los requerimientos del usuario.

La Figura 17.2 muestra que el desarrollo y el prototipado incremental tienen objetivos diferentes:

1. El objetivo del desarrollo incremental es entregar a los usuarios finales un sistema funcional. Esto significa que normalmente debe comenzar con los requerimientos del usuario que mejor se comprendan y que tengan la prioridad más alta. Los requerimientos inciertos y de prioridad más baja se implementan cuando sean requeridos por los usuarios.
2. El objetivo del prototipado desechable es validar u obtener los requerimientos del sistema. Debe comenzar con aquellos requerimientos que no se comprendan bien ya que requiere saber más de ellos. Puede no necesitar nunca prototipar los requerimientos sencillos.

Otra diferencia importante entre estos enfoques está en la gestión de la calidad de los sistemas. Los prototipos desechables tienen un periodo de vida muy corto. Debe ser posible cambiarlos rápidamente durante el desarrollo, pero no se requiere mantenimiento a largo plazo. En dicho prototipo se puede permitir un rendimiento pobre y una baja fiabilidad siempre y cuando ayude a entender los requerimientos.

Por el contrario, los sistemas desarrollados incrementalmente en que las versiones iniciales evolucionan a un sistema final se deben desarrollar con los mismos estándares de calidad de la organización que cualquier otro software. Deben tener una estructura robusta para que se les pueda dar mantenimiento durante muchos años. Deben ser fiables y eficientes, y deben estar acordes con los estándares organizacionales apropiados.

17.1 Métodos ágiles

En los años 80 y principios de los 90, existía una opinión general de que la mejor forma de obtener un mejor software era a través de una planificación cuidadosa del proyecto, una garantía de calidad formalizada, la utilización de métodos de análisis y diseño soportados por herramientas CASE, y procesos de desarrollo de software controlados y rigurosos. Esta opinión provenía, fundamentalmente, de la comunidad de ingenieros de software implicada en el desarrollo de grandes sistemas software de larga vida que normalmente se componían de un gran número de programas individuales.

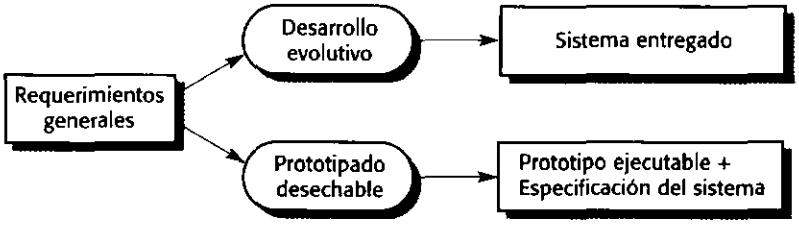


Figura 17.2 Desarrollo y prototipado incremental.

Algunos o la totalidad de estos programas eran a menudo sistemas críticos, como se indicó en el Capítulo 3. Este software era desarrollado por grandes equipos que a veces trabajaban para compañías diferentes. A menudo estaban dispersos geográficamente y trabajaban en el software durante largos periodos de tiempo. Un ejemplo de este tipo de software son los sistemas de control de un avión moderno, en los cuales pueden transcurrir hasta 10 años desde la especificación inicial hasta la utilización. Estos enfoques, algunos de los cuales trato en este libro, implican una importante sobrecarga de trabajo en cuanto a la planificación, diseño y documentación del sistema. Este esfuerzo adicional se justifica cuando se tiene que coordinar el trabajo de múltiples equipos de desarrollo, cuando el sistema es un sistema crítico y cuando muchas personas diferentes estarán involucradas en el mantenimiento del software durante su vida.

Sin embargo, cuando este enfoque «pesado» de desarrollo basado en la planificación fue aplicado a sistemas de negocio pequeños y de tamaño medio, el esfuerzo invertido era tan grande que algunas veces dominaba el proceso de desarrollo del software. Se pasaba más tiempo pensando en cómo se debía desarrollar el sistema que en programar el desarrollo y las pruebas. Cuando cambiaban los requerimientos, se hacía esencial rehacer el trabajo, y al menos en principio, la especificación y el diseño tenían que cambiar con el programa.

El descontento con estos enfoques pesados condujo a varios desarrolladores de software en los años 90 a proponer nuevos métodos ágiles. Éstos permitieron a los equipos de desarrollo centrarse en el software mismo en vez de en su diseño y documentación. Los métodos ágiles universalmente dependen de un enfoque iterativo para la especificación, desarrollo y entrega del software, y principalmente fueron diseñados para apoyar al desarrollo de aplicaciones de negocio donde los requerimientos del sistema normalmente cambiaban rápidamente durante el proceso de desarrollo. Están pensados para entregar software funcional de forma rápida a los clientes, quienes pueden entonces proponer que se incluyan en iteraciones posteriores del sistema nuevos requerimientos o cambios en los mismos.

Probablemente el método ágil más conocido es la programación extrema (Beck, 1999; Beck, 2000), que se describe posteriormente en este capítulo. Sin embargo, otros enfoques ágiles son Scrum (Schwaber y Beedle, 2001), Cristal (Cockburn, 2001), Desarrollo de Software Adaptable (Highsmith, 2000), DSDM (Stapleton, 1997) y Desarrollo Dirigido por Características (Palmer y Felsing, 2002). El éxito de estos métodos ha llevado a una cierta integración con métodos de desarrollo más tradicionales basados en el modelado de sistemas, dando por resultado la noción de modelado ágil (Ambler y Jeffries, 2002) y las instanciaciones ágiles del Proceso Unificado de Rational (Larman, 2002).

Aunque todos estos métodos ágiles se basan en la noción de desarrollo y entrega incrementales, proponen procesos diferentes para alcanzarla. Sin embargo, comparten un conjunto de principios y, por lo tanto, tienen mucho en común. En la Figura 17.3 se muestran estos principios.

Los partidarios de los métodos ágiles han sido evangélicos en la promoción de su uso y han tendido a pasar por alto sus deficiencias. Esto ha provocado una respuesta igualmente radical, la cual, exagera los problemas de este enfoque (Stephens y Rosenberg, 2003). Críticos más razonables como DeMarco y Boehm (DeMarco y Boehm, 2002) resaltan tanto las ventajas como las desventajas de los métodos ágiles. Proponen que un enfoque híbrido en el cual los métodos ágiles incorporen algunas técnicas del desarrollo basado en la planificación puede ser lo mejor a largo plazo.

En la práctica, sin embargo, los principios subyacentes a los métodos ágiles son a veces difíciles de realizar:

Principio	Descripción
Participación del cliente	Los clientes deben estar fuertemente implicados en todo el proceso de desarrollo. Su papel es proporcionar y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del sistema.
Entrega incremental	El software se desarrolla en incrementos, donde el cliente especifica los requerimientos a incluir en cada incremento.
Personas, no procesos	Se deben reconocer y explotar las habilidades del equipo de desarrollo. Se les debe dejar desarrollar sus propias formas de trabajar, sin procesos formales, a los miembros del equipo.
Aceptar el cambio	Se debe contar con que los requerimientos del sistema cambian, por lo que el sistema se diseña para dar cabida a estos cambios.
Mantener la simplicidad	Se deben centrar en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Donde sea posible, se trabaja activamente para eliminar la complejidad del sistema.

Figura 17.3
Los principios de los métodos ágiles.

1. Si bien la idea de la participación del cliente en el proceso de desarrollo es atractiva, su éxito depende de tener un cliente que esté dispuesto y pueda pasar tiempo con el equipo de desarrollo y que pueda representar a todos los stakeholders del sistema. Frecuentemente, los representantes de los clientes están sometidos a otras presiones y no pueden participar plenamente en el desarrollo del software.
2. Los miembros individuales del equipo pueden no tener la personalidad apropiada para la participación intensa que es típica de los métodos ágiles. Por lo tanto, es posible que no se relacionen adecuadamente con los otros miembros del equipo.
3. Priorizar los cambios puede ser extremadamente difícil, especialmente en sistemas en lo que existen muchos stakeholders. Por lo general, cada stakeholders proporciona prioridades distintas a diferentes cambios.
4. Mantener la simplicidad requiere un trabajo extra. Bajo presión por las agendas de entregas, los miembros del equipo pueden no tener tiempo de llevar a cabo las simplificaciones deseables del sistema.

Otro problema no técnico, el cual es un problema general con el desarrollo y entrega incrementales, ocurre cuando los clientes del sistema utilizan una organización externa para el desarrollo del sistema. Como se explicó en el Capítulo 6, normalmente el documento de requerimientos del software es parte del contrato entre el cliente y el proveedor. Puesto que la especificación incremental es inherente a los métodos ágiles, redactar contratos para este tipo de desarrollo puede ser difícil.

Por consiguiente, los métodos ágiles tienen que depender de contratos donde el cliente paga por el tiempo necesario para el desarrollo del sistema en vez de por el desarrollo de un conjunto de requerimientos específico. Siempre y cuando vaya todo bien, esto beneficia tanto al cliente como al desarrollador. Sin embargo, si surgen problemas puede haber disputas sobre quién es el culpable y quién debe pagar por el tiempo extra y recursos necesarios para resolver los problemas.

Todos los métodos tienen límites, y los métodos ágiles son sólo apropiados para algunos tipos de desarrollo de sistemas. Son los más idóneos para el desarrollo de sistemas de negocio pequeños y de tamaño medio y para el desarrollo de productos para ordenadores perso-

nales. No son adecuados para el desarrollo de sistemas a gran escala con equipos de desarrollo ubicados en diferentes lugares y donde puedan haber complejas interacciones con otros sistemas hardware o software. No se deben utilizar los métodos ágiles para el desarrollo de sistemas críticos en los que es necesario un análisis detallado de todos los requerimientos del sistema para comprender sus implicaciones de seguridad o protección.

17.2 Programación extrema

La Programación Extrema (XP) es posiblemente el método ágil más conocido y ampliamente utilizado. El nombre fue acuñado por Beck (Beck, 2000) debido a que el enfoque fue desarrollado utilizando buenas prácticas reconocidas, como el desarrollo iterativo, y con la participación del cliente en niveles «extremos».

En la programación extrema, todos los requerimientos se expresan como escenarios (llamados historias de usuario), los cuales se implementan directamente como una serie de tareas. Los programadores trabajan en parejas y desarrollan pruebas para cada tarea antes de escribir el código. Todas las pruebas se deben ejecutar satisfactoriamente cuando el código nuevo se integre al sistema. Existe un pequeño espacio de tiempo entre las entregas del sistema. La Figura 17.4 ilustra el proceso de la XP para producir un incremento del sistema que se está desarrollando.

La programación extrema implica varias prácticas, resumidas en la Figura 17.5, que se ajustan a los principios de los métodos ágiles:

1. El desarrollo incremental se lleva a cabo través de entregas del sistema pequeñas y frecuentes y por medio de un enfoque para la descripción de requerimientos basado en las historias de cliente o escenarios que pueden ser la base para el proceso de planificación.
2. La participación del cliente se lleva a cabo a través del compromiso a tiempo completo del cliente en el equipo de desarrollo. Los representantes de los clientes participan en el desarrollo y son los responsables de definir las pruebas de aceptación del sistema.
3. El interés en las personas, en vez de en los procesos, se lleva a cabo a través de la programación en parejas, la propiedad colectiva del código del sistema, y un proceso de desarrollo sostenible que no implique excesivas jornadas de trabajo.
4. El cambio se lleva a cabo a través de las entregas regulares del sistema, un desarrollo previamente probado y la integración continua.
5. El mantenimiento de la simplicidad se lleva a cabo a través de la refactorización constante para mejorar la calidad del código y la utilización de diseños sencillos que no prevén cambios futuros en el sistema.

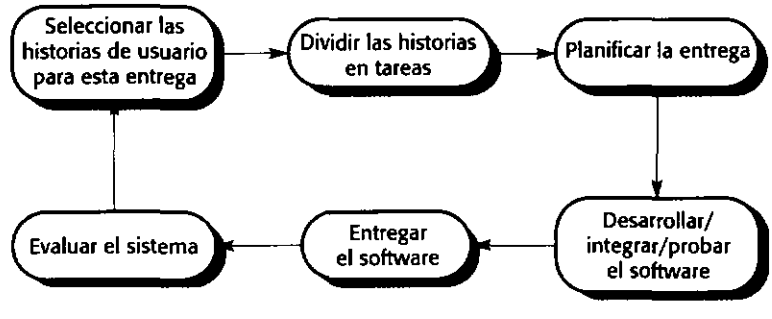


Figura 17.4 El ciclo de entrega en la programación extrema.

Principio o práctica	Descripción
Planificación incremental	Los requerimientos se registran en tarjetas de historias y las historias a incluir en una entrega se determinan según el tiempo disponible y su prioridad relativa. Los desarrolladores dividen estas Historias en «Tareas» de desarrollo. Véanse las Figuras 17.6 y 17.7.
Entregas pequeñas	El mínimo conjunto útil de funcionalidad que proporcione valor de negocio se desarrolla primero. Las entregas del sistema son frecuentes e incrementalmente añaden funcionalidad a la primera entrega.
Diseño sencillo	Sólo se lleva a cabo el diseño necesario para cumplir los requerimientos actuales.
Desarrollo previamente probado	Se utiliza un sistema de pruebas de unidad automatizado para escribir pruebas para nuevas funcionalidades antes de que éstas se implementen.
Refactorización	Se espera que todos los desarrolladores refactoricen el código continuamente tan pronto como encuentren posibles mejoras en el código. Esto conserva el código sencillo y mantenible.
Programación en parejas	Los desarrolladores trabajan en parejas, verificando cada uno el trabajo del otro y proporcionando la ayuda necesaria para hacer siempre un buen trabajo.
Propiedad colectiva	Las parejas de desarrolladores trabajan en todas las áreas del sistema, de modo que no desarrollen islas de conocimientos y todos los desarrolladores posean todo el código. Cualquiera puede cambiar cualquier cosa.
Integración continua	En cuanto acaba el trabajo en una tarea, se integra en el sistema entero. Después de la integración, se deben pasar al sistema todas las pruebas de unidad.
Ritmo sostenible	No se consideran aceptables grandes cantidades de horas extras, ya que a menudo el efecto que tienen es que se reduce la calidad del código y la productividad a medio plazo.
Cliente presente	Debe estar disponible al equipo de la XP un representante de los usuarios finales del sistema (el cliente) a tiempo completo. En un proceso de la programación extrema, el cliente es miembro del equipo de desarrollo y es responsable de formular al equipo los requerimientos del sistema para su implementación.

Figura 17.5
Prácticas de la programación extrema.

En un proceso de la XP, los clientes están fuertemente implicados en la especificación y establecimiento de prioridades de los requerimientos del sistema. Los requerimientos no se especifican como una lista de funciones requeridas del sistema. Más bien, los clientes del sistema son parte del equipo de desarrollo y discuten escenarios con otros miembros del equipo. Desarrollan conjuntamente una «tarjeta de historias» (*story card*) que recoge las necesidades del cliente. El equipo de desarrollo intentará entonces implementar ese escenario en una entrega futura del software. En la Figura 17.6 se ilustra un ejemplo de una tarjeta de historia para el sistema LIBSYS, basada en un escenario del Capítulo 7.

Descarga e impresión de un artículo

En primer lugar, seleccione el artículo que desea de una lista visualizada. Tiene entonces que decirle al sistema cómo lo pagará –se puede hacer a través de una suscripción, una cuenta de empresa o mediante una tarjeta de crédito.

Después de esto, obtiene un formulario de derechos de autor del sistema para que lo rellene. Cuando lo haya enviado, se descarga el artículo en su computadora.

Elija una impresora y se imprimirá una copia del artículo. Le dice al sistema que la impresión se ha realizado correctamente.

Si es un artículo de sólo impresión, no puede guardar la versión en PDF, por lo que automáticamente se elimina de su computadora.

Figura 17.6 Tarjeta de historia para la descarga de documentos.

Una vez que se han desarrollado las tarjetas de historias, el equipo de desarrollo las divide en tareas y estima el esfuerzo y recursos requeridos para su implementación. El cliente establece entonces la prioridad de las historias a implementar, eligiendo aquellas historias que pueden ser utilizadas inmediatamente para entregar un apoyo útil al negocio. Por supuesto, cuando los requerimientos cambian, las historias sin implementar también cambian o se pueden descartar. Si se requieren cambios en un sistema que ya se ha entregado, se desarrollan nuevas tarjetas de historias y, de nuevo, el cliente decide si estos cambios tienen prioridad sobre nuevas funcionalidades.

La programación extrema adopta un enfoque «extremo» para el desarrollo iterativo. Se pueden construir varias veces al día nuevas versiones del software y los incrementos se entregan al cliente cada dos meses aproximadamente. Cuando un programador construye el sistema para crear una versión nueva, debe ejecutar todas las pruebas automatizadas existentes además de las pruebas para las funcionalidades nuevas. El nuevo software generado solamente se acepta si se ejecutan satisfactoriamente todas las pruebas.

Un precepto fundamental de la ingeniería del software tradicional es que se debe diseñar para el cambio. Esto es, hay que prever los cambios futuros en el software y diseñar éste de forma que tales cambios se puedan implementar fácilmente. Sin embargo, la programación extrema ha descartado este principio partiendo del hecho de que diseñar para el cambio es a menudo un esfuerzo inútil. Con frecuencia los cambios previstos nunca se materializan y realmente se efectúan peticiones de cambios completamente diferentes.

El problema con la implementación de cambios imprevistos es que tienden a degradar la estructura del software, por lo que los cambios se hacen cada vez más difíciles de implementar. La programación extrema aborda este problema sugiriendo que se debe refactorizar constantemente el software. Esto significa que el equipo de programación busca posibles mejoras del software y las implementa inmediatamente. Por lo tanto, el software siempre debe ser fácil de entender y cambiar cuando se implementen nuevas historias.

17.2.1 Pruebas en XP

Como se ha indicado en la introducción de este capítulo, una de las diferencias importantes entre el desarrollo iterativo y el desarrollo basado en la planificación es la forma de probar el sistema. Con el desarrollo iterativo, no existe una especificación del sistema que pueda ser utilizada por un equipo de pruebas externo para desarrollar las pruebas del sistema. Por consi-

guiente, algunos enfoques para el desarrollo iterativo tienen un proceso de pruebas muy informal.

Para evitar algunos de los problemas de las pruebas y de la validación del sistema, XP pone más énfasis en el proceso de pruebas que otros métodos ágiles. Las pruebas del sistema son fundamentales en XP, en la que se ha desarrollado un enfoque que reduce la probabilidad de producir nuevos incrementos del sistema que introduzcan errores en el software existente.

Las características clave de las pruebas en XP son:

1. Desarrollo previamente probado.
2. Desarrollo de pruebas incremental a partir de los escenarios.
3. Participación del usuario en el desarrollo de las pruebas y en la validación.
4. El uso de bancos de pruebas automatizados.

El desarrollo previamente probado es una de las innovaciones más importantes en XP. Al escribir primero las pruebas se define implícitamente tanto una interfaz como una especificación del funcionamiento para la funcionalidad a desarrollar. Se reducen los problemas en los requerimientos y las confusiones de la interfaz. Se puede adoptar este enfoque en cualquier proceso en el que exista una relación clara entre un requerimiento del sistema y el código que implementa ese requerimiento. En XP, siempre puede verse este vínculo debido a que las tarjetas de historias que representan los requerimientos se dividen en tareas, y las tareas son las unidades principales de implementación.

Como se ha señalado, los requerimientos de usuario en XP se expresan como escenarios o historias y el usuario establece las prioridades de éstos para su desarrollo. El equipo de desarrollo evalúa cada escenario y lo divide en tareas. Cada tarea representa una característica distinta del sistema y se puede diseñar entonces una prueba de unidad para esa tarea. Por ejemplo, en la Figura 17.7 se muestran algunas de las tarjetas de tareas (*task cards*) desarrolladas a partir de la tarjeta de historia para la descarga de documentos (Figura 17.6).

Cada tarea genera una o más pruebas de unidad que verifican la implementación descrita en esa tarea. Por ejemplo, la Figura 17.8 es una descripción abreviada de un caso de pruebas que se ha desarrollado para comprobar que se ha implementado un número de tarjeta de crédito válido.

El papel del cliente en el proceso de pruebas es ayudar a desarrollar las pruebas de aceptación para las historias que se tienen que implementar en la siguiente entrega del sistema. Como se explica en el Capítulo 23, las pruebas de aceptación son el proceso en el que se prueba el sistema utilizando datos del cliente para verificar que cumple sus necesidades reales.

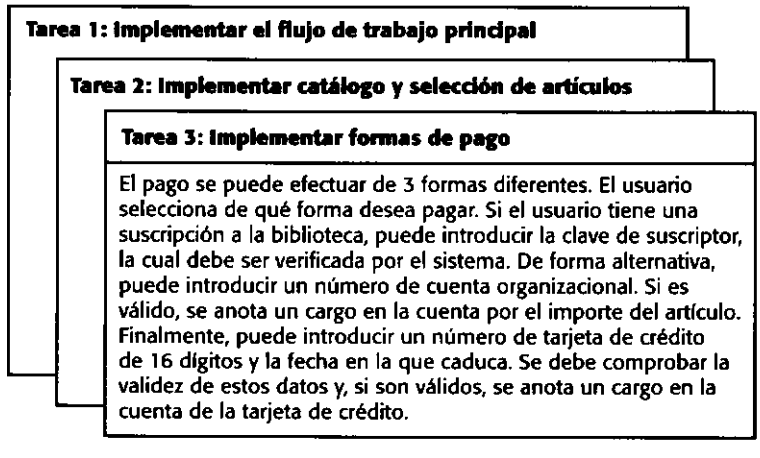


Figura 17.7 Tarjetas de tareas para la descarga de documentos.

Prueba 4: Prueba de la validez de la tarjeta de crédito
Entrada: Una cadena que representa el número de tarjeta de crédito y dos enteros que representan el mes y el año de la caducidad de la tarjeta. **Pruebas:** Comprobar que todos los bytes de la cadena son dígitos. Comprobar que el mes se encuentra entre 1 y 12 y que el año es mayor o igual que el año actual. Utilizando los 4 primeros dígitos del número de tarjeta de crédito, comprobar que el emisor de la tarjeta es válido consultando la tabla de emisores de tarjetas. Comprobar la validez de la tarjeta de crédito enviando el número de tarjeta y la fecha en la que caduca el emisor de la tarjeta. **Salida:** OK o un mensaje de error indicando que la tarjeta no es válida.

Figura 17.8
Descripción del caso de prueba para comprobar la validez de la tarjeta de crédito.

En XP, las pruebas de aceptación, como el desarrollo, son incrementales. Para esta historia en particular, la prueba de aceptación implicaría seleccionar varios documentos, pagarlos de diferentes formas e imprimirlos en impresoras distintas. En la práctica, probablemente se desarrollaría una serie de pruebas de aceptación en vez de una única prueba.

El desarrollo previamente probado y el uso de bancos de pruebas automatizados son las principales virtudes del enfoque de la XP. En vez de escribir el código del programa y luego las pruebas de ese código, el desarrollo previamente probado significa que las pruebas se escriben antes que el código. Fundamentalmente, las pruebas se escriben como un componente ejecutable antes de que se implemente la tarea. Una vez que se ha implementado el software, se pueden ejecutar las pruebas inmediatamente. Este componente de pruebas debe ser una aplicación independiente, debe simular el envío de la entrada a probar y debe verificar que el resultado cumple la especificación de salida. El banco de pruebas automatizado es un sistema que envía a ejecución estas pruebas automatizadas.

Con el desarrollo previamente probado, siempre hay un conjunto de pruebas que se pueden ejecutar fácilmente y de forma rápida. Esto significa que siempre que se añada cualquier funcionalidad al sistema, se pueden ejecutar las pruebas y detectar inmediatamente los problemas que el código nuevo haya introducido.

En el desarrollo previamente probado, las personas encargadas de implementar las tareas tienen que comprender perfectamente la especificación, de modo que puedan escribir las pruebas del sistema. Esto significa que se tienen que clarificar las ambigüedades y omisiones en la especificación antes de que empiece la implementación. Además, esto también evita el problema del «retraso de las pruebas», en el que, debido a que los desarrolladores del sistema trabajan a un ritmo más elevado que los probadores, cada vez se adelanta más la implementación sobre las pruebas y hay una tendencia a saltarse las pruebas para poder mantener la agenda establecida.

Sin embargo, el desarrollo previamente probado no siempre funciona según lo previsto. Los programadores prefieren la programación a las pruebas y algunas veces escriben pruebas incompletas que no comprueban las situaciones excepcionales. Además, puede ser difícil escribir algunas pruebas. Por ejemplo, en una interfaz de usuario compleja, con frecuencia es difícil escribir las pruebas de unidad para el código que implementa la «vista lógica» y el flujo de trabajo entre las pantallas. Por último, es difícil juzgar la completitud de un conjunto de pruebas. Aunque se pueda tener una gran cantidad de pruebas del sistema, ese conjunto de pruebas puede no proporcionar una cobertura completa. Es posible que no se ejecuten partes cruciales del sistema y, por lo tanto, se queden sin probar.

Contar con el cliente para el apoyo al desarrollo de las pruebas de aceptación es a veces un problema serio en el proceso de pruebas de la XP. Las personas que adoptan el papel de cliente tienen muy poco tiempo disponible y es posible que no puedan trabajar a tiempo completo con el equipo de desarrollo. El cliente puede pensar que proporcionar los requerimientos es contribución suficiente y puede ser reacio a participar en el proceso de pruebas.

17.2.2 Programación en parejas

Otra práctica innovadora que se ha introducido es que los programadores trabajan en parejas para desarrollar el software. De hecho, se sientan juntos en la misma estación de trabajo para desarrollar el software. El desarrollo no siempre implica la misma pareja de personas trabajando juntas. Más bien, la idea es que las parejas se creen de forma dinámica para que todos los miembros del equipo puedan trabajar con los otros miembros en una pareja de programación durante el proceso de desarrollo.

El uso de la programación en parejas tiene varias ventajas:

1. Apoya la idea de la propiedad y responsabilidad comunes del sistema. Esto refleja la idea de Weinberg de la programación sin ego (Weinberg, 1971), en la que el equipo como un todo es dueño del software y las personas individuales no tienen la culpa de los problemas con el código. En cambio, el equipo tiene una responsabilidad colectiva para resolver estos problemas.
2. Actúa como un proceso de revisión informal ya que cada línea de código es vista por al menos dos personas. Las inspecciones y revisiones del código (tratadas en el Capítulo 22) consiguen descubrir un alto porcentaje de errores del software. Sin embargo, requiere mucho tiempo organizarlas y, por lo general, generan retrasos en el proceso de desarrollo. A pesar de que la programación en parejas es un proceso menos formal que probablemente no encuentre tantos errores, es un proceso de inspección mucho más económico que las inspecciones formales de programas.
3. Ayuda en la refactorización, la cual es un proceso de mejora del software. Un principio de la XP es que se debe refactorizar constantemente el software. Es decir, se deben escribir nuevamente partes del código para mejorar su claridad y estructura. El problema para implementar esto en un entorno de desarrollo normal es que es un esfuerzo que se gasta para obtener un beneficio a largo plazo, y se puede juzgar a una persona que practique la refactorización como menos eficiente que otra que simplemente realice el desarrollo del código. Cuando se utiliza la programación en parejas y la propiedad colectiva, otros se benefician inmediatamente con la refactorización, por lo que probablemente apoyarán el proceso.

Podría pensarse que la programación en parejas es menos eficiente que la programación individual y que una pareja de desarrolladores produciría, como mucho, la mitad del código que dos personas trabajando individualmente. Sin embargo, diversos estudios sobre desarrollos que utilizan XP no confirman esto. La productividad del desarrollo con programación en parejas parece ser comparable con la de dos personas trabajando de forma independiente (Williams *et al.*, 2000). Las razones para esto son que las parejas discuten sobre el software antes de empezar el desarrollo, por lo probablemente tengan menos comienzos en falso y tengan que rehacer menos trabajo, y que el número de errores evitados debido a la inspección informal es tal que se pasa menos tiempo arreglando errores descubiertos durante el proceso de pruebas.

17.3 Desarrollo rápido de aplicaciones

Aunque los métodos ágiles como un enfoque para el desarrollo iterativo han recibido una gran atención en los últimos años, los sistemas de negocio se han desarrollado de forma iterativa durante muchos años utilizando técnicas de desarrollo rápido de aplicaciones. Las técnicas de desarrollo rápido de aplicaciones (RAD) evolucionaron de los llamados lenguajes de cuarta generación en los años 80 y se utilizan para desarrollar aplicaciones con un uso intensivo de datos. Por consiguiente, normalmente están organizadas como un conjunto de herramientas que permiten crear datos, buscarlos, visualizarlos y presentarlos en informes. La Figura 17.9 ilustra una organización típica de un sistema RAD.

Las herramientas que se incluyen en un entorno RAD son:

1. *Un lenguaje de programación de bases de datos*, que contiene conocimiento de la estructura de la base de datos y que incluye las operaciones básicas de manipulación de bases de datos. El lenguaje estándar de programación de base de datos es SQL (Groff *et al.*, 2002). Los comandos SQL se pueden introducir directamente o generar de forma automática a partir de formularios rellenados por los usuarios finales.
2. *Un generador de interfaces*, que se utiliza para crear formularios de introducción y visualización de datos.
3. *Enlaces a aplicaciones de oficina*, como una hoja de cálculo para el análisis y manipulación de información numérica o un procesador de textos para la creación de plantillas de informes.
4. *Un generador de informes*, que se utiliza para definir y crear informes a partir de la información de la base de datos.

Los sistemas RAD tienen éxito debido a que, como se explicó en el Capítulo 13, las aplicaciones de negocio tienen muchas cosas en común. Esencialmente, a menudo estas aplicaciones comprenden la actualización de una base de datos y la producción de informes a partir de la información existente en ella. Se utilizan formularios estándar para las entradas y salidas. Los sistemas RAD están dirigidos a la producción de aplicaciones interactivas que se apoyan la abstracción de la información en una base de datos organizacional, presentándola a los usuarios finales en su terminal o estación de trabajo, y actualizando la base de datos con los cambios hechos por los usuarios.

Muchas de las aplicaciones de negocio se apoyan en formularios estructurados para las entradas y salidas, por lo que los entornos RAD proporcionan recursos potentes para la definición de pantallas y generación de informes. A menudo, las pantallas se definen como una se-

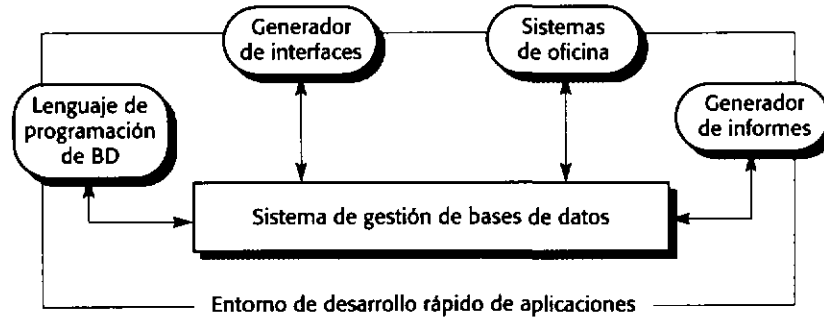


Figura 17.9
Un entorno de desarrollo rápido de aplicaciones.

rie de formularios vinculados (en una aplicación que estudiamos, hubo 137 definiciones de formularios), por lo que el sistema de generación de pantallas debe proporcionar:

1. *Definición de formularios interactivos* que permitan al desarrollador definir los campos a visualizar y la manera en que éstos deben organizarse.
2. *Vinculación de los formularios* que permitan al desarrollador especificar que ciertas entradas provocan la visualización de formularios adicionales.
3. *Verificación de campos* que permitan al desarrollador definir los rangos permitidos para los valores de entrada en los campos de los formularios.

Hoy en día, muchos entornos RAD permiten el desarrollo de interfaces de bases de datos basadas en navegadores web. Dichos entornos permiten el acceso a la base de datos desde cualquier lugar a través de una conexión válida de Internet. Esto reduce los costes de formación y de software, y permite a los usuarios externos tener acceso a una base de datos. Sin embargo, las limitaciones inherentes de los navegadores web y los protocolos de Internet implican que este enfoque no sea adecuado para sistemas en los que se requieran respuestas interactivas muy rápidas.

Actualmente, la mayoría de los sistemas RAD incluyen herramientas de programación visual que permiten desarrollar sistemas de forma interactiva. En vez de escribir un programa secuencial, el desarrollador del sistema manipula iconos gráficos que representan funciones, datos o componentes de interfaces de usuario, y asocia el procesamiento de secuencias de comandos con estos iconos. Se genera automáticamente un programa ejecutable a partir de la representación visual del sistema.

Los sistemas de desarrollo visual, como Visual Basic, permiten este enfoque basado en la reutilización para el desarrollo de aplicaciones. Los programadores de éstas construyen el sistema de forma interactiva definiendo la interfaz en términos de pantallas, campos, botones y menús. A éstos, se les asigna un nombre y se asocia el procesamiento de secuencias de comandos con partes individuales de la interfaz (por ejemplo, un botón denominado Simular). Estas secuencias de comandos pueden hacer llamadas a componentes reutilizables, a código de propósito especial o a una mezcla de ambos.

Este enfoque se ilustra en la Figura 17.10, la cual muestra una pantalla de aplicación que incluye menús en la parte superior, campos de entrada (los campos blancos a la izquierda de la pantalla), campos de salida (el campo gris a la izquierda de la pantalla) y botones (los rectángulos redondeados a la derecha de la pantalla). Cuando el sistema de programación visual ubica en la pantalla estas entidades, el desarrollador define qué componentes reutilizables se deben asociar con ellos o escribe un fragmento de programa para llevar a cabo el procesamiento requerido. También se muestran en la Figura 17.10 los componentes asociados con algunos de los elementos de la pantalla.

Visual Basic es un ejemplo muy sofisticado de un lenguaje de creación de secuencias de comandos (Ousterhout, 1998). Éstos son lenguajes de alto nivel sin tipos diseñados que ayudan a integrar componentes para crear sistemas. Uno de los primeros lenguajes de este tipo fue el shell de Unix (Bourne, 1978; Gordon y Bieman, 1995); desde su desarrollo, se han creado varios lenguajes de creación de secuencias de comandos más potentes (Ousterhout, 1994; Lutz, 1996; Wall *et al.*, 1996). Estos lenguajes incluyen estructuras de control y juegos de herramientas gráficas que, como ilustra Ousterhout (Ousterhout, 1998), pueden reducir radicalmente el tiempo requerido para el desarrollo del sistema.

Este enfoque para el desarrollo de sistemas permite el desarrollo rápido de aplicaciones relativamente sencillas que pueden ser construidas por un equipo pequeño de personas. Es más difícil de organizar para sistemas más grandes que deben ser desarrollados por equipos con

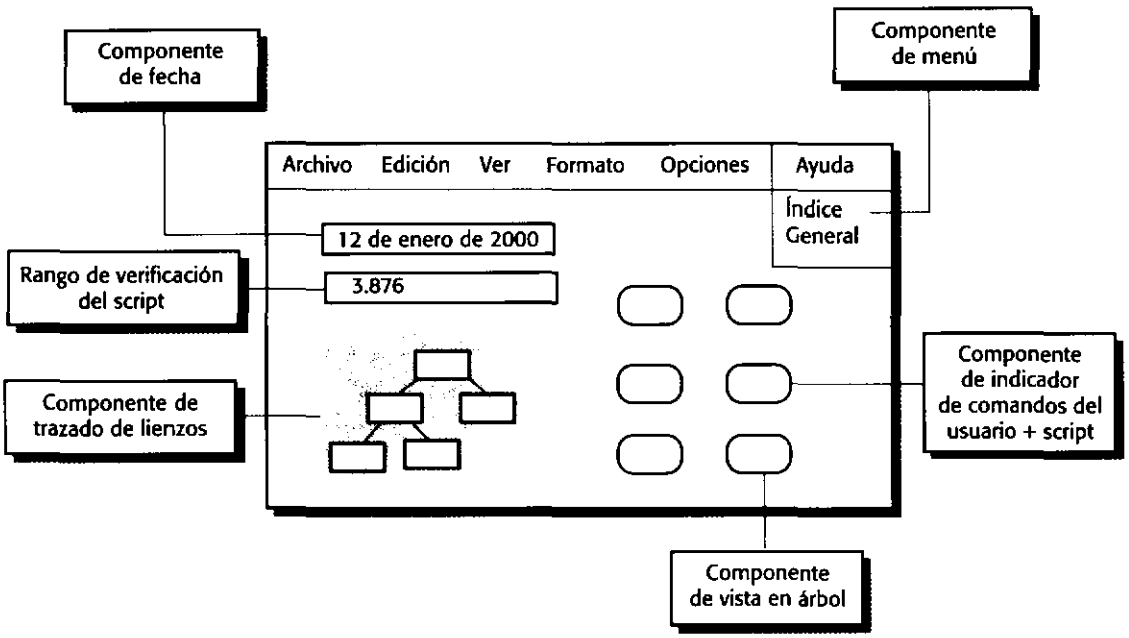


Figura 17.10 Programación visual con reutilización.

un mayor número de personas. No existe una arquitectura explícita del sistema y a menudo existen dependencias complejas entre las partes del sistema, lo que puede causar problemas cuando se requieran cambios. Además, debido a que los lenguajes de creación de secuencias de comandos se limitan a un conjunto específico de objetos en interacción, puede ser difícil implementar interfaces de usuario no estándares.

El *desarrollo visual* es un enfoque a RAD que utiliza la integración de componentes software reutilizables de grano fino. Un enfoque alternativo basado en la reutilización reutiliza «componentes» que son sistemas de aplicaciones completos. Éste se denomina a veces desarrollo basado en COTS, donde COTS (Commercial Off-the-Shelf) significa que las aplicaciones ya están disponibles. Por ejemplo, si un sistema requiere las funcionalidades de un procesador de textos, podría utilizar un procesador de textos estándar como Microsoft Word. En el Capítulo 18 se estudia el desarrollo basado en COTS desde la perspectiva de la reutilización.

Para ilustrar el tipo de aplicación que se podría desarrollar utilizando un enfoque basado en COTS, consideremos el proceso de gestión de requerimientos descrito en el Capítulo 7. Un sistema de apoyo a dicha gestión necesita una forma de capturar y almacenar los requerimientos, producir los informes, descubrir las relaciones entre los requerimientos y gestionar estas relaciones como tablas de rastreo. En un enfoque basado en COTS, se podría construir un prototipo vinculando una base de datos (para almacenar los requerimientos), un procesador de textos (para capturar los requerimientos y los formatos de los informes), una hoja de cálculo (para gestionar las tablas de rastreo) y código escrito específicamente para encontrar las relaciones entre los requerimientos.

El desarrollo basado en COTS da acceso al desarrollador a toda la funcionalidad de una aplicación. Si ésta también proporciona recursos de creación de secuencias de comandos o de ajuste (por ejemplo, macros de Excel), éstos se pueden utilizar para desarrollar cierto tipo de funcionalidad de la aplicación. Para comprender este enfoque de desarrollo de aplicaciones

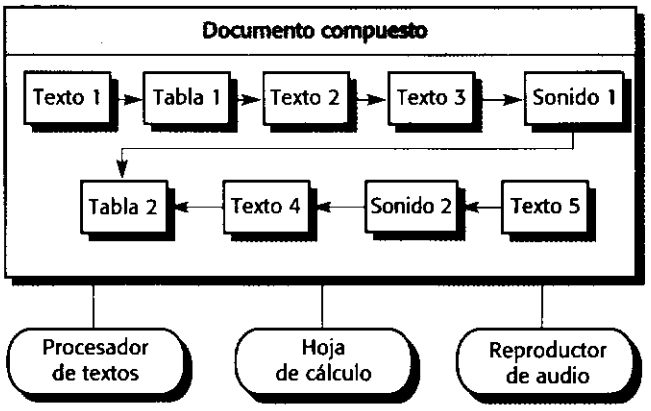


Figura 17.11 Vinculación de aplicaciones.

es de utilidad una metáfora de composición de documentos. Los datos procesados por el sistema se pueden organizar en un documento compuesto que actúa como un contenedor de varios objetos. Estos objetos contienen diferentes tipos de datos (como una tabla, un diagrama, un formulario) que pueden ser procesados por diferentes aplicaciones. Los objetos se enlazan y se clasifican para que al acceder a un objeto se inicie la aplicación asociada.

La Figura 17.11 ilustra un sistema de aplicaciones formado por un documento compuesto que incluye elementos de texto, de hojas de cálculo y archivos de sonido. Los elementos de texto son procesados por el procesador de textos; las tablas, por la aplicación de hojas de cálculo, y los archivos de sonido, por el reproductor de audio. Cuando un usuario del sistema accede a un objeto de un tipo particular, se llama a la aplicación asociada para proporcionar la funcionalidad al usuario. Por ejemplo, cuando se accede a objetos de tipo sonido, se llama al reproductor de audio para procesarlos.

La principal ventaja de este enfoque es que mucha de la funcionalidad de la aplicación se puede implementar rápidamente a un coste muy bajo. Los usuarios que ya estén familiarizados con las aplicaciones que componen el sistema no tendrán que aprender cómo utilizar las nuevas características. Sin embargo, si no saben cómo utilizar las aplicaciones, el aprendizaje puede ser difícil, especialmente si están confundidos con la funcionalidad innecesaria de la aplicación. Puede haber también problemas de rendimiento en la aplicación debido a la necesidad de cambiar de una aplicación del sistema a otra. Este esfuerzo adicional para realizar el cambio entre aplicaciones depende de la ayuda que proporcione el sistema operativo.

17.4 Prototipado del software

Como se ha indicado en la introducción de este capítulo, existen algunas circunstancias en las que, por razones prácticas o contractuales, no se puede utilizar un proceso de entrega del software incremental. En esas situaciones, se completa una declaración de los requerimientos del sistema y es utilizada por el equipo de desarrollo como la base para el software del sistema. Como se ha explicado, se pueden conseguir algunos de los beneficios de un proceso de desarrollo incremental creando un prototipo del software. Este enfoque se denomina a veces prototipado desechable debido a que el prototipo no es entregado al cliente o mantenido por el desarrollador.

Un prototipo es una versión inicial de un sistema software que se utiliza para demostrar conceptos, probar opciones de diseño y, en general, informarse más del problema y sus posi-

bles soluciones. El desarrollo rápido e iterativo del prototipo es esencial, de modo que los costes sean controlados y los stakeholders del sistema puedan experimentar con el prototipo en las primeras etapas del proceso del software.

Un prototipo del software se puede utilizar de varias maneras en un proceso de desarrollo de software:

1. En el proceso de ingeniería de requerimientos, un prototipo puede ayudar en la obtención y validación de los requerimientos del sistema.
2. En el proceso de diseño del sistema, se puede utilizar un prototipo para explorar soluciones software particulares y para apoyar al diseño de las interfaces de usuario.
3. En el proceso de pruebas, se puede utilizar un prototipo para ejecutar pruebas back-to-back con el sistema que se entregarán al cliente.

Los prototipos del sistema permiten a los usuarios ver cómo éste apoya su trabajo. Pueden adquirir nuevas ideas para los requerimientos y encontrar áreas fuertes y débiles en el software. Entonces pueden proponer nuevos requerimientos del sistema. Además, a medida que se desarrolla el prototipo, puede revelar errores y omisiones en los requerimientos propuestos. Una función descrita en una especificación podría parecer útil y bien definida. Sin embargo, cuando la función se combina con otras, a menudo los usuarios comprueban que su visión inicial fue incorrecta o incompleta. La especificación del sistema podría modificarse para reflejar el cambio en la comprensión de los requerimientos.

Se puede utilizar un prototipo del sistema mientras se esté diseñando el sistema para llevar a cabo experimentos de diseño con el fin de verificar la viabilidad de un diseño propuesto. Por ejemplo, un diseño de una base de datos puede ser prototipado y probado para verificar que las consultas más comunes de los usuarios tienen el acceso a los datos más eficiente. El prototipado es también una parte fundamental del proceso de diseño de las interfaces de usuario. Debido a la naturaleza dinámica de las interfaces de usuario, las descripciones textuales y los diagramas no son suficientes para expresar los requerimientos de éstas. Por lo tanto, el prototipado rápido con la participación del usuario final es la única forma razonable de desarrollar interfaces gráficas de usuario para sistemas software.

Un problema importante en las pruebas del sistema es la validación de las pruebas, donde tiene que comprobarse si los resultados de una prueba son lo que se esperaba. Cuando está disponible un prototipo del sistema, se puede reducir el esfuerzo realizado en la comprobación de los resultados ejecutando pruebas back-to-back (Figura 17.12). Se envían los mismos casos de prueba tanto al prototipo como al sistema en prueba. Si ambos dan el mismo resultado, probablemente el caso de prueba no haya detectado ningún defecto. Si los resultados difieren, puede significar que hay un defecto en el sistema y se deben investigar las razones de la diferencia.

Por último, además de apoyar las actividades del proceso del software, se pueden utilizar prototipos a fin de reducir el tiempo requerido para desarrollar la documentación del usuario y para formarlos. Un sistema funcional, aunque limitado, está disponible de forma rápida para demostrar la viabilidad y utilidad de la aplicación a la dirección.

En un estudio de 39 proyectos de prototipado, Gordon y Bieman (Gordon y Bieman, 1995) observaron que los beneficios de utilizar el prototipado fueron:

1. Mejora en la usabilidad del sistema
2. Una mejor concordancia entre el sistema y las necesidades del usuario
3. Mejora en la calidad del diseño
4. Mejora en el mantenimiento
5. Reducción en el esfuerzo de desarrollo

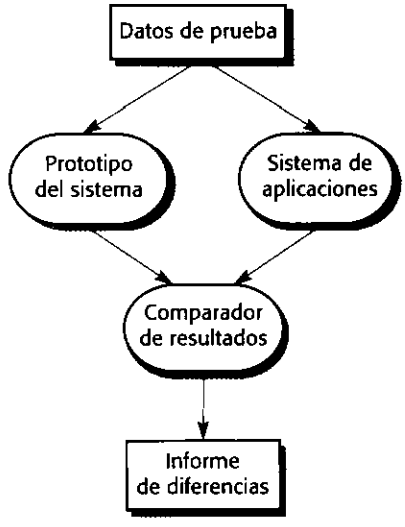


Figura 17.12
Pruebas back-to-back.

Su estudio sugiere que las mejoras en la usabilidad y en la definición de requerimientos del usuario debido a la utilización de un prototipo no significan necesariamente un incremento general en los costes de desarrollo. Por lo general, la construcción de prototipos incrementa los costes en las etapas iniciales del proceso del software pero reduce los costes posteriores en el proceso de desarrollo. La razón principal de esto es que se evita rehacer el trabajo durante el desarrollo debido a que los clientes solicitan menos cambios en el sistema. Sin embargo, Gordon y Bieman observaron que el rendimiento general del sistema algunas veces se degrada si se reutiliza código ineficiente proveniente del prototipo.

En la Figura 17.13 se muestra un modelo del proceso para el desarrollo de prototipos. Los objetivos de la construcción de éstos deben ser explícitos desde el inicio del proceso. Éstos pueden ser desarrollar un sistema para construir un prototipo de la interfaz de usuario, desarrollar un sistema para validar los requerimientos funcionales del sistema o para demostrar la viabilidad de la aplicación a la dirección. El mismo prototipo no puede cumplir todos los objetivos. Si éstos no se especifican, la dirección o los usuarios finales pueden malinterpretar la función del prototipo. En consecuencia, es posible que no obtengan los beneficios que esperan del desarrollo de éste.

La siguiente etapa en el proceso es decidir qué incluir y, quizás lo más importante, qué excluir del sistema prototipo. Para reducir los costes de la construcción del prototipo y acelerar la agenda de entregas, se puede excluir de éste cierta funcionalidad. Se puede decidir relajar los requerimientos no funcionales, como el tiempo de respuesta y la utilización de la memoria. La gestión y manejo de errores se puede pasar por alto o hacerse de forma rudimentaria,

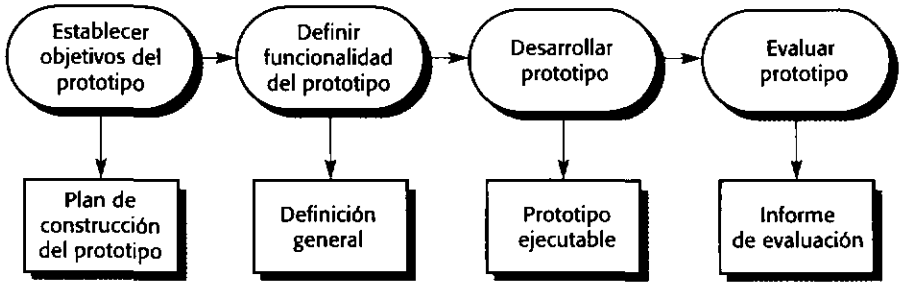


Figura 17.13
El proceso de desarrollo de prototipos.

a menos que el objetivo del prototipo sea establecer una interfaz de usuario. Se pueden reducir los estándares de fiabilidad y calidad de la programación.

La etapa final del proceso es la evaluación del prototipo. En ésta se debe prever la formación del usuario y se deben utilizar los objetivos del prototipo para obtener un plan de evaluación. Los usuarios requieren tiempo para acostumbrarse a un nuevo sistema y utilizarlo de forma normal. Una vez que lo utilizan, descubren los errores y omisiones en los requerimientos.

Un problema general con el desarrollo de un prototipo desechable ejecutable es que el modo de utilizarlo puede que no se corresponda con el modo en que se utiliza el sistema final entregado. El probador del prototipo puede no ser el típico usuario de éste. El tiempo de formación durante la evaluación del prototipo puede ser insuficiente. Si el prototipo es lento, los evaluadores pueden modificar su forma de trabajar y evitar aquellas características que tengan tiempos de respuesta lentos. Si el sistema final tiene mejor tiempo de respuesta, pueden utilizarlo de forma diferente.

Algunas veces, los gerentes presionan a los desarrolladores para que entreguen los prototipos desechables, especialmente cuando existen retrasos en la entrega de la versión final del software. En vez de hacer frente a los retrasos en el proyecto, los gerentes pueden creer que entregar un sistema incompleto o de baja calidad es mejor que nada. Sin embargo, normalmente esto no es aconsejable por las siguientes razones:

1. Puede ser imposible ajustar el prototipo para que cumpla con los requerimientos no funcionales que fueron dejados de lado durante su desarrollo, como los de rendimiento, protección, robustez y fiabilidad.
2. El cambio rápido durante el desarrollo significa, inevitablemente, que no se documenta el prototipo. La única especificación del diseño es el código del prototipo. Esto no es suficiente para el mantenimiento a largo plazo.
3. Los cambios hechos durante el desarrollo del prototipo probablemente degradan la estructura del sistema. Éste será difícil y caro de mantener.
4. Los estándares de calidad organizacionales normalmente se relajan para el desarrollo del prototipo.

Los prototipos desechables no tienen que ser ejecutables para ser de utilidad en el proceso de ingeniería de requerimientos. Como se explica en el Capítulo 16, las maquetas en papel de la interfaz de usuario (Rettig, 1994) pueden ser efectivas para ayudar a los usuarios a perfeccionar un diseño de la interfaz y a trabajar a través de escenarios de utilización. Éstos son muy baratos de desarrollar y se pueden construir en pocos días. Una extensión de esta técnica es un prototipo Mago de Oz en el que sólo se desarrolla la interfaz de usuario. Los usuarios interactúan con esta interfaz, pero sus peticiones se pasan a una persona que los interpreta y muestra la respuesta apropiada.

PUNTOS CLAVE

- **Al crecer la presión por una entrega rápida del software, se utiliza cada vez más un enfoque iterativo para el desarrollo del software como una técnica de desarrollo estándar para sistemas pequeños y de tamaño medio, especialmente en el dominio de los negocios.**

- Los métodos ágiles son métodos de desarrollo iterativo que se centran en la especificación, diseño e implementación del sistema de forma incremental. Implican directamente a los usuarios en el proceso de desarrollo. Reducir la sobrecarga en cuanto al esfuerzo de desarrollo puede hacer posible un desarrollo del software más rápido.
- La programación extrema es un método ágil conocido que integra una variedad de buenas prácticas de programación, como las pruebas sistemáticas, la continua mejora del software y la participación del cliente en el equipo de desarrollo.
- Un punto fuerte particular de la programación extrema es el desarrollo de pruebas automatizadas antes de que se cree una funcionalidad en un programa. Se deben ejecutar de forma satisfactoria todas las pruebas cuando se integra un incremento en un sistema.
- El desarrollo rápido de aplicaciones implica la utilización de entornos de desarrollo que incluyan herramientas potentes para apoyar la producción del sistema. Éstas comprenden lenguajes de programación de bases de datos, generadores de formularios e informes, y enlaces a aplicaciones de oficina.
- El prototipado desechable es un proceso de desarrollo iterativo en el que se utiliza un sistema prototipo para explorar los requerimientos y las opciones de diseño. Este prototipo no está destinado para su utilización por parte de los clientes del sistema.
- Cuando se implementa un prototipo desechable, primero se tienen que desarrollar las partes del sistema que menos se comprenden; por el contrario, en un enfoque de desarrollo incremental, se empieza desarrollando las partes del sistema que mejor se comprenden.

LECTURAS ADICIONALES

Extreme Programming Explained. Éste fue el primer libro sobre XP y es todavía, quizás, el que más merece la pena leer. Explica el enfoque desde la perspectiva de uno de sus inventores, y llega muy claramente su entusiasmo. (Kent Beck, 2000, Addison-Wesley.)

«Get ready for agile methods, with care». Una seria crítica de los métodos ágiles que analiza sus puntos fuertes y débiles, redactada por ingenieros de software tremendamente experimentados. (B. Boehm, *IEEE Computer,* enero de 2002.)

«Scripting: Higher-level programming for the 21st century». Una visión general de los lenguajes de creación de secuencias de comandos por el inventor de Tcl/Tk, quien describe las ventajas de este enfoque para el desarrollo rápido de aplicaciones. (J. K. Ousterhout, *IEEE Computer,* marzo de 1998.)

DSDM: Dynamic Systems Development Meted. Una descripción de un enfoque para el desarrollo rápido de aplicaciones que algunas personas consideran que es un ejemplo inicial de un método ágil. (J. Stapleton, 1997, Addison-Wesley.)

EJERCICIOS

17.1 Explique por qué la entrega y desarrollo rápidos de sistemas nuevos es a menudo más importante para las empresas que una funcionalidad detallada de estos sistemas.

17.2 Explique cómo los principios subyacentes a los métodos ágiles conducen a un desarrollo y utilización del software acelerados.

17.3 ¿Cuándo recomendaría *contra* el uso de un método ágil para desarrollar un sistema software?

17.4 La programación extrema expresa los requerimientos del usuario como historias, donde cada historia se redacta en una tarjeta. Comente las ventajas y desventajas de este enfoque para la descripción de requerimientos.

17.5 Explique por qué el desarrollo previamente probado ayuda al programador a desarrollar una mejor comprensión de los requerimientos del sistema. ¿Cuáles son los problemas potenciales del desarrollo previamente probado?

17.6 Sugiera cuatro razones por las que el índice de productividad de programadores trabajando en parejas sea aproximadamente el mismo que dos programadores trabajando de forma individual.

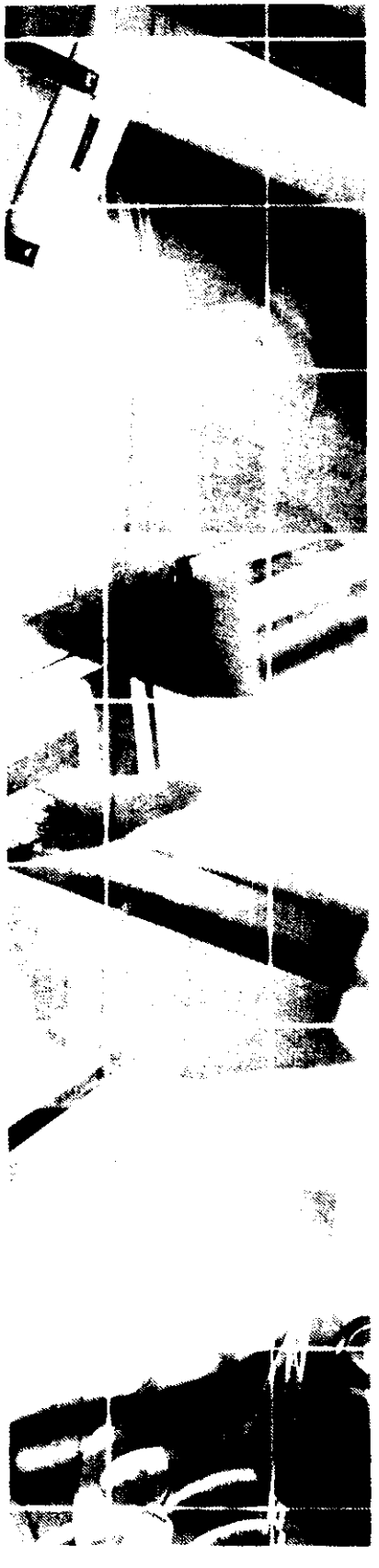
17.7 Se le ha pedido que investigue la viabilidad de la construcción de prototipos en el proceso de desarrollo de software de su organización. Redacte un informe para su jefe que explique las clases de proyectos donde se deba utilizar el prototipado, y precise los costes y beneficios esperados de éste.

17.8 Un administrador de software trabaja en el desarrollo de un proyecto de diseño de un sistema de ayuda que traduce los requerimientos del software a una especificación software formal. Comente las ventajas y desventajas de las siguientes estrategias de desarrollo:

a) Desarrollo de un prototipo desechable, su evaluación y posterior revisión de los requerimientos del sistema. Desarrollo del sistema final utilizando C.
b) Desarrollo del sistema a partir de los requerimientos existentes utilizando Java, posterior modificación para que se adapte a cualquier cambio en los requerimientos del usuario.
c) Desarrollo del sistema utilizando un desarrollo incremental con la participación del cliente en el equipo de desarrollo.

17.9 Una organización benéfica le ha solicitado construir un prototipo de un sistema que dé seguimiento al historial de todas las donaciones que ha recibido. Este sistema tiene que almacenar los nombres y direcciones de los donantes, sus intereses particulares, la cantidad donada y cuándo fue donada. Si la donación está por encima de cierta cantidad, el donante puede poner condiciones para hacerla (por ejemplo, debe gastarse en un proyecto en particular), y el sistema debe mantener el historial de éstas y cómo fueron gastadas. Comente cómo construiría el prototipo de este sistema, teniendo presente que la organización tiene una mezcla de trabajadores asalariados y voluntarios. Muchos de los voluntarios son jubilados que tienen poca o nula experiencia en informática.

17.10 Usted ha desarrollado un prototipo desechable de un sistema para un cliente que está muy contento con él. Sin embargo, sugiere que no existe necesidad de desarrollar otro sistema, sino que debe entregarle el prototipo y le ofrece un precio excelente por él. Usted sabe que habrá problemas futuros en el mantenimiento del sistema. Explique cómo debería responder a este cliente.



18 Reutilización del software

Objetivos

Los objetivos de este capítulo son introducir la reutilización del software y explicar cómo la reutilización contribuye al proceso de desarrollo del software. Cuando haya leído este capítulo:

- comprenderá los beneficios y los problemas de reutilizar software cuando se desarrollan sistemas nuevos;
- habrá aprendido varias formas de implementar la reutilización del software;
- comprenderá el concepto de reutilización y cómo los conceptos reutilizables se pueden representar como patrones o ser embebidos en generadores de programas;
- habrá aprendido cómo los sistemas pueden desarrollarse rápidamente mediante la composición de grandes aplicaciones comerciales;
- habrá sido introducido en las líneas de productos software que están formadas por una arquitectura común y componentes configurables y reutilizables.

Contenidos

El proceso de diseño en la mayoría de las disciplinas de ingeniería se basa en la reutilización de sistemas o componentes existentes. Los ingenieros mecánicos o eléctricos no especifican normalmente un diseño en el que cada componente tenga que ser fabricado de una forma especial. Basan su diseño en componentes que han sido utilizados y probados en otros sistemas. Estos componentes no son solamente pequeños componentes, como tuercas y válvulas sino que incluyen subsistemas mayores, como motores, condensadores o turbinas.

La ingeniería del software basada en reutilización es una estrategia de ingeniería del software comparable en la que el proceso de desarrollo es adaptado a la reutilización de software existente. Si bien los beneficios de la reutilización han sido reconocidos durante muchos años (McIlroy, 1968), sólo en la última década ha existido una transición gradual desde el desarrollo del software original hasta el desarrollo basado en reutilización. La tendencia hacia el desarrollo basado en reutilización viene dada como respuesta a las demandas de una menor producción de software y de menores costes de mantenimiento, de una entrega más rápida de los sistemas y del incremento en la calidad del software. Cada vez más compañías ven su software como un activo valioso y están promocionando la reutilización para incrementar sus beneficios en las inversiones de software.

La ingeniería del software basada en reutilización es una aproximación del desarrollo que intenta maximizar la reutilización del software existente. Las unidades de software que se reutilizan pueden ser de tamaños totalmente diferentes. Por ejemplo:

1. *Reutilización de sistemas de aplicaciones.* La totalidad de un sistema de aplicaciones puede ser reutilizada incorporándolo sin ningún cambio en otros sistemas, configurando la aplicación para diferentes clientes o desarrollando familias de aplicaciones que tienen una arquitectura común pero que son adaptadas a clientes particulares. La reutilización de sistemas de aplicaciones se trata en la Sección 18.5.
2. *Reutilización de componentes.* La reutilización de componentes de una aplicación varía en tamaño desde subsistemas hasta objetos simples. Por ejemplo, un sistema de emparejamiento de patrones desarrollado como parte de un sistema de procesamiento de textos puede ser reutilizado en un sistema de gestión de base de datos. Esto se trata en el Capítulo 19.
3. *Reutilización de objetos y funciones.* Pueden reutilizarse componentes software que implementan una única función, como por ejemplo una función matemática o una clase de objetos. Esta forma de reutilización, basada en librerías estándar, ha sido habitual en los cuarenta últimos años. Están disponibles muchas librerías de funciones y clases para diferentes tipos de aplicaciones y plataformas de desarrollo. Éstas pueden utilizarse fácilmente enlazándolas con código de otras aplicaciones. En áreas como algoritmos matemáticos y gráficos, donde se necesita a un experto específico para desarrollar objetos y funciones, ésta es una aproximación particularmente efectiva.

Los sistemas y componentes software son entidades reutilizables específicas, pero su naturaleza específica significa a veces que el coste de modificarlos para una nueva situación resulta elevado. Una forma complementaria de reutilización es la *reutilización de conceptos*, en la que, en lugar de reutilizar un componente, la entidad reutilizada es más abstracta y se diseña para ser configurada y adaptada a una variedad de situaciones. La reutilización de conceptos puede incluirse en aproximaciones tales como patrones de diseño, productos de sistemas configurables y generadores de programas. El proceso de reutilización, cuando se reutilizan los conceptos, incluye una actividad de instanciación en la que los conceptos abstractos se configuran para una situación concreta. Estas dos aproximaciones de reutilización de conceptos —patrones de diseño y generación de programas— se tratan más adelante en este capítulo.

La ventaja obvia de la reutilización de software es que los costes totales de desarrollo deberían reducirse. Se necesita especificar, diseñar, implementar y validar menos componentes software. Sin embargo, la reducción de costes es sólo una ventaja de la reutilización. En la Figura 18.1, se muestran otras ventajas de la reutilización de activos software.

Sin embargo, hay también costes y problemas asociados con la reutilización (Figura 18.2). En particular, hay un coste significativo asociado con el estudio de si un componente es apropiado para su reutilización en una situación concreta y con la prueba de ese componente para asegurar su confiabilidad. Estos costes adicionales pueden inhibir la introducción de la reutilización y puede implicar que las reducciones de los costes totales de desarrollo mediante reutilización sean menores que las de los costes anticipados.

La reutilización sistemática no se lleva a cabo sin más, sino que debe ser planificada e introducida ampliamente en una organización a través de un programa de reutilización. Esto ha sido reconocido durante muchos años en Japón (Matsumoto, 1984), en donde la reutilización es una parte integral de la aproximación de «factoría» japonesa al desarrollo del software (Cusamano, 1989). Compañías como Hewlett-Packard han experimentado con éxito la reutilización de programas (Griss y Wosser, 1995), y su experiencia ha sido incorporada en un libro de propósito general por Jacobsen y otros (Jacobsen *et al.*, 1997).

Beneficio	Explicación
Incremento de la confiabilidad	El software reutilizado, que ha sido usado y probado en sistemas en funcionamiento, debería ser más confiable que el software nuevo debido a que sus fallos en la implementación y el diseño ya han sido encontrados y reparados.
Reducción del riesgo del proceso	El coste del software existente ya es conocido, mientras que el coste de desarrollo es siempre cuestionable. Éste es un factor importante para la gestión de proyectos debido a que reduce el margen de error en la estimación de costes del proyecto. Esto es particularmente cierto cuando se reutilizan componentes software relativamente grandes tales como subsistemas.
Uso efectivo de especialistas	En lugar de hacer el mismo trabajo una y otra vez, estos especialistas de aplicaciones pueden desarrollar software reutilizable que encapsule su conocimiento.
Cumplimiento de estándares	Algunos estándares, tales como los estándares de interfaz de usuario, pueden implementarse como un conjunto de componentes reutilizables estándar. Por ejemplo, si los menús en un interfaz de usuario se implementan usando componentes reutilizables, todas las aplicaciones presentan el mismo formato de menú a los usuarios. El uso de interfaces de usuario estándares mejora la confiabilidad debido a que es menos probable que los usuarios cometan errores cuando se encuentran con una interfaz familiar.
Desarrollo acelerado	Sacar al mercado un sistema tan pronto como sea posible es a menudo más importante que los costes totales de desarrollo. La reutilización del software puede acelerar la producción del sistema debido a que se reducen los tiempos de desarrollo y validación.

Figura 18.1 Beneficios de la reutilización de software.

Problema	Explicación
Incremento en los costes de mantenimiento	Si el código fuente de un sistema de software reutilizado o un componente no está disponible, entonces los costes de mantenimiento pueden incrementarse debido a que los elementos reutilizados del sistema son cada vez más incompatibles con los cambios del sistema.
Falta de soporte de las herramientas	Los conjuntos de herramientas CASE no soportan el desarrollo con reutilización. Puede ser difícil o imposible integrar estas herramientas con un sistema de librería de componentes. El proceso del software asumido por estas herramientas puede no tener en cuenta la reutilización.
Síndrome «reinventar la rueda»	Algunos ingenieros software prefieren reescribir componentes debido a que creen que pueden mejorarlos. Esto es en parte cierto ya que la escritura original de software es vista como un reto mayor que la utilización de software de otras personas.
Creación y mantenimiento de una librería de componentes	Puede ser caro construir una librería de componentes reutilizable y asegurar que los desarrolladores de software puedan usarla. Las técnicas actuales para clasificar, catalogar y recuperar componentes software son todavía inmaduras.
Búsqueda, comprensión y adaptación de componentes reutilizables	Los componentes software tienen que buscarse en una librería, entenderse y, algunas veces, adaptarse al trabajo en un entorno nuevo. Los ingenieros deben confiar razonablemente en que van a encontrar un componente en la librería antes de que puedan incluir la búsqueda de un componente como parte de su proceso normal de desarrollo.

Figura 18.2 Problemas con la reutilización.

18.1 El campo de la reutilización

Durante los veinte últimos años, se han desarrollado muchas técnicas para soportar la reutilización del software. Éstas explotan el hecho de que los sistemas del mismo dominio de aplicación son similares y tienen potencial para la reutilización. Esta reutilización es posible a diferentes niveles (desde funciones simples a aplicaciones completas), y los estándares para componentes reutilizables facilitan la reutilización. La Figura 18.3 muestra varias formas de soportar la reutilización del software, cada una de las cuales se describe brevemente en la Figura 18.4.

Dada esta lista de técnicas para reutilización, la cuestión clave es ¿cuál es la técnica más adecuada a utilizar? Obviamente, esto depende de los requerimientos del sistema a desarrollar, la tecnología y activos reutilizables disponibles, y la experiencia del grupo de desarrollo. Los factores clave que deberían considerarse a la hora de planificar la reutilización son:

1. *La agenda de desarrollo del software*. Si el software tiene que desarrollarse rápidamente, debería intentarse reutilizar sistemas comerciales en vez de componentes individuales. Éstos son activos reutilizables de grano grueso. Si bien el cumplimiento de los requerimientos puede no ser perfecto, esta aproximación minimiza la cantidad de desarrollo requerido.
2. *Vida esperada del software*. Si se está desarrollando un sistema de larga vida, habría que centrarse en la mantenibilidad del sistema. En esas circunstancias, no solamente

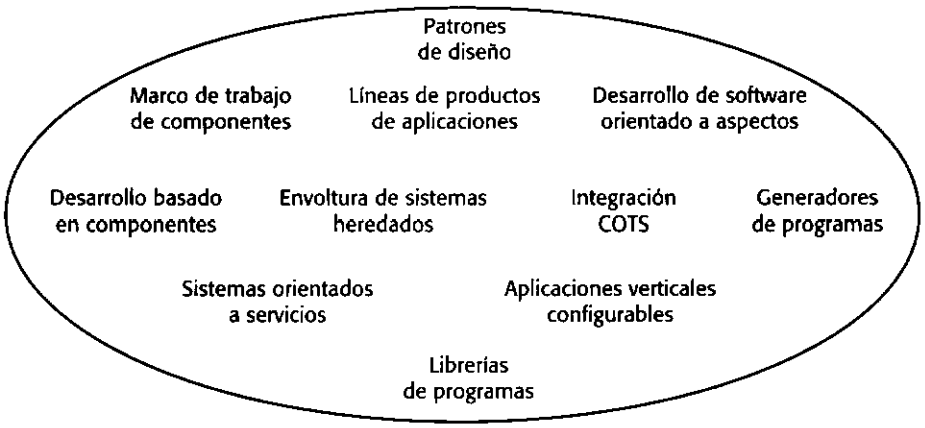


Figura 18.3
El campo de la reutilización.

Aproximación	Descripción
Patrones de diseño	Las abstracciones genéricas similares entre aplicaciones se representan como patrones de diseño que muestran los objetos abstractos y concretos y sus interacciones.
Desarrollo basado en componentes	Los sistemas se desarrollan integrando componentes (colecciones de objetos) que cumplen los estándares de modelado de componentes. Esto se trata en el Capítulo 19.
Marcos de aplicaciones	Las colecciones de clases concretas y abstractas pueden adaptarse y extenderse para crear sistemas de aplicaciones.
Envoltura de sistemas heredados	Sistemas heredados (véase el Capítulo 2) que pueden ser «envueltos» definiendo un conjunto de interfaces y proporcionando acceso a estos sistemas heredados a través de estas interfaces.
Sistemas orientados a servicios	Los sistemas se desarrollan enlazando servicios compartidos, que pueden ser proporcionados de forma externa.
Líneas de productos de aplicaciones	Un tipo de aplicación se generaliza alrededor de una arquitectura común para que pueda ser adaptada para diferentes clientes.
Integración COTS	Los sistemas se desarrollan integrando sistemas de aplicaciones existentes.
Aplicaciones verticales configurables	Un sistema genérico se diseña para que pueda configurarse para las necesidades de clientes de sistemas particulares.
Librerías de programas	Están disponibles para reutilización las librerías de funciones y de clases que implementan abstracciones comúnmente usadas.
Generadores de programas	Un sistema generador incluye conocimiento de un tipo de aplicación particular y puede generar sistemas o fragmentos de un sistema en ese dominio.
Desarrollo del software orientado a aspectos	Componentes compartidos entrelazados en una aplicación en diferentes lugares cuando se compila el programa.

Figura 18.4
Aproximaciones que soportan la reutilización del software.

debería pensarse en las posibilidades inmediatas de la reutilización, sino también en las implicaciones a largo plazo. Se tendrá que adaptar el sistema a nuevos requerimientos, lo cual probablemente signifique hacer cambios a los componentes y cómo éstos son utilizados. Si no se tiene acceso al código fuente, probablemente debería evitarse utilizar componentes y sistemas de proveedores externos; no se puede estar seguro de que estos proveedores serán capaces de continuar soportando el software reutilizado.

3. *Los conocimientos, habilidades y experiencia del grupo de desarrollo.* Todas las tecnologías de reutilización son bastante complejas y se necesita bastante tiempo para comprenderlas y usarlas de forma efectiva. Sin embargo, si el grupo de desarrollo posee habilidades en un área particular, probablemente habría que centrarse en ella.
4. *La criticidad del software y sus requerimientos no funcionales.* Para un sistema crítico que tiene que ser certificado por un regulador externo, se tiene que crear un caso de confiabilidad para el sistema (tratado en el Capítulo 24). Esto es difícil si no se tiene acceso al código fuente del software. Si el software tiene requerimientos de rendimiento estrictos, puede ser imposible utilizar estrategias tales como reutilización a través de generadores de programas. Estos sistemas tienden a generar código relativamente ineficiente.
5. *El dominio de las aplicaciones.* En algunos dominios de aplicaciones como los sistemas de información médica y de fabricación, hay varios productos genéricos que pueden reutilizarse para configurarlos a una situación particular. Si se está trabajando en tales dominios, siempre deberían considerarse éstos como una opción.
6. *La plataforma sobre la que el sistema se va a ejecutar.* Algunos modelos de componentes, como COM/Active X, son plataformas específicas de Microsoft. Si se está desarrollando sobre una plataforma como éstas, esta aproximación puede ser la más adecuada. De igual forma, los sistemas de aplicaciones genéricos pueden ser plataformas específicas y solamente es posible reutilizarlos si el sistema se diseña para la misma plataforma.

La cantidad de sistemas de reutilización disponibles es tal que, en la mayoría de las situaciones, existe la posibilidad de alguna reutilización del software. Si se consigue o no la reutilización es a menudo una cuestión de gestión más que una cuestión técnica. Los gestores pueden ser reacios a comprometer sus requerimientos para permitir el uso de componentes reutilizables, o pueden decidir que el desarrollo de componentes originales podría ayudar a crear una base de activos software. Es posible que no comprendan los riesgos asociados con la reutilización tan bien como entienden los riesgos del desarrollo original. Sin embargo, si bien los riesgos de un nuevo desarrollo software pueden ser más elevados, algunos gestores pueden preferir los riesgos conocidos a los desconocidos.

18.2 Patrones de diseño

Cuando el diseñador intenta reutilizar componentes ejecutables, está limitado de forma inevitable por las decisiones de diseño detallado que han sido tomadas por los implementadores de esos componentes. Éstas varían desde algoritmos particulares que han sido utilizados para implementar los componentes hasta los objetos y tipos en las interfaces de los componentes. Cuando estas decisiones de diseño están en pugna con sus requerimientos particulares, reutilizar el componente es imposible o introduce ineficiencias en su sistema.

Una forma de solventar esto es reutilizar diseños abstractos que no incluyen detalles de la implementación. El diseñador puede implementarlos para ajustarse a sus requerimientos particulares de la aplicación. Las primeras instancias de esta aproximación para reutilización aparecieron en la documentación y publicación de algoritmos fundamentales (Knuth, 1971) y, más tarde, en la documentación de tipos abstractos de datos tales como pilas, árboles y listas (Booch, 1987). Más recientemente, esta aproximación para la reutilización ha sido incluida en los patrones de diseño.

Los patrones de diseño se derivaron de las ideas introducidas por Christopher Alexander (Alexander *et al.*, 1977), quien sugirió que existían ciertos patrones del diseño de edificios que eran comunes e inherentemente interesantes y efectivos. El patrón es una descripción del problema y la esencia de su solución, de forma que la solución se pueda reutilizar en diferentes situaciones. El patrón no es una especificación detallada. Antes bien, puede pensarse en él como una descripción del conocimiento y experiencia acumulados. Es una solución adecuada a un problema común. Una frase del sitio web hillside.net, que se dedica a mantener información sobre patrones, encapsula su papel en la reutilización:

> *Los patrones y los lenguajes de patrones son formas de describir las mejores prácticas, buenos diseños, y encapsulan la experiencia de tal forma que es posible para otros el reutilizar dicha experiencia.*

La mayoría de los diseñadores piensan en los patrones de diseño como una forma de soportar el diseño orientado a objetos. Los patrones a menudo tienen en cuenta características de los objetos tales como la herencia y el polimorfismo para proporcionar generalidad. Sin embargo, el principio general de encapsulación de experiencia en un patrón es que es igualmente aplicable a todas las aproximaciones de diseño software.

Gamma y otros (Gamma *et al.*, 1995) definen los cuatro elementos esenciales de los patrones de diseño:

1. Un nombre que es una referencia significativa del patrón.
2. Una descripción del área del problema que explica cuándo puede aplicarse el patrón.
3. Una descripción de las partes de la solución del diseño, sus relaciones y sus responsabilidades. Ésta no es una descripción concreta de diseño. Es una plantilla para una solución de diseño que puede instanciarse de diferentes formas. A menudo ésta se expresa gráficamente y muestra las relaciones entre los objetos y las clases de los objetos en la solución.
4. Una declaración de las consecuencias —los resultados y compromisos— de aplicar el patrón. Esto puede ayudar a los diseñadores a comprender si un patrón puede ser aplicado de forma efectiva en una situación particular.

Estos elementos esenciales de la descripción de un patrón pueden descomponerse, tal y como se muestra en el ejemplo de la Figura 18.5. Por ejemplo, Gamma y sus coautores dividen la descripción del problema en motivación (una descripción de por qué el patrón es útil) y aplicabilidad (una descripción de las situaciones en las que el patrón puede utilizarse). Bajo la descripción de la solución, ellos describen la estructura del patrón, participantes, colaboraciones e implementación.

Para ilustrar la descripción de patrones, utilizamos el patrón Observer, tomado del libro de Gamma y otros. Este patrón puede utilizarse en una gran variedad de situaciones en las que se requieren diferentes presentaciones del estado de un objeto. Separa el objeto que debe ser visualizado de las diferentes formas de presentación. Esto se ilustra en la Figura 18.6, que muestra dos presentaciones gráficas del mismo conjunto de datos. En la descripción, se utili-

Nombre del patrón: Observer

Descripción: Separa la vista del estado de un objeto del objeto mismo y permite proporcionar vistas alternativas. Cuando el estado del objeto cambia, todas las vistas son notificadas y actualizadas de forma automática para reflejar el cambio.

Descripción del problema: En muchas situaciones es necesario proporcionar múltiples vistas de alguna información del estado, tal como una vista gráfica y una vista tabular. No todos ellos pueden ser conocidos cuando se especifica la información. Todas las presentaciones alternativas pueden soportar interacción y, cuando el estado se cambia, deben actualizarse todas las vistas.

Este patrón puede utilizarse en todas las situaciones en las que se requiera más de un formato de vista para la información del estado y en las que no es necesario que el objeto que mantenga la información del estado conozca los formatos específicos utilizados para las vistas.

Descripción de la solución: La estructura del patrón se muestra en la Figura 18.7. Ésta define dos objetos abstractos, Subject y Observer, y dos objetos concretos, ConcreteSubject y ConcreteObserver, que heredan los atributos de los objetos abstractos relacionados. El estado a visualizar es mantenido en ConcreteSubject, que también hereda las operaciones de Subject permitiéndole añadir y eliminar Observers y enviar una notificación cuando el estado ha cambiado.

El objeto ConcreteObserver mantiene una copia del estado de ConcreteSubject e implementa la interfaz Update () de Observer, que permite guardar estas copias en el mismo momento. El objeto ConcreteObserver visualiza automáticamente su estado —esto no es generalmente una operación de interfaz.

Consecuencias: El objeto Subject solamente conoce el Observer abstracto y no conoce los detalles de la clase concreta. Por ello hay un mínimo acoplamiento entre estos objetos. Debido a esta falta de conocimiento, las optimizaciones que mejoran el rendimiento de la visualización no son prácticas. Los cambios en el objeto Subject pueden provocar la generación de un conjunto de actualizaciones vinculadas con los objetos Observers, algunas de las cuales pueden no ser necesarias.

Figura 18.5
Una descripción del patrón Observer.

zan las descripciones de los cuatro elementos fundamentales y los complemento con una breve declaración de lo que el patrón puede hacer.

Las representaciones gráficas se utilizan normalmente para ilustrar las clases de objetos que se usan en patrones y sus relaciones. Esta descripción del patrón complementa y añade detalles a la descripción de la solución. La Figura 18.7 es la representación en UML del patrón Observer.

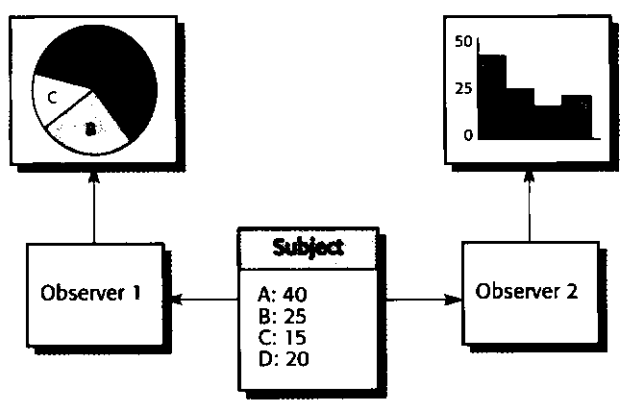


Figura 18.6
Múltiples vistas.

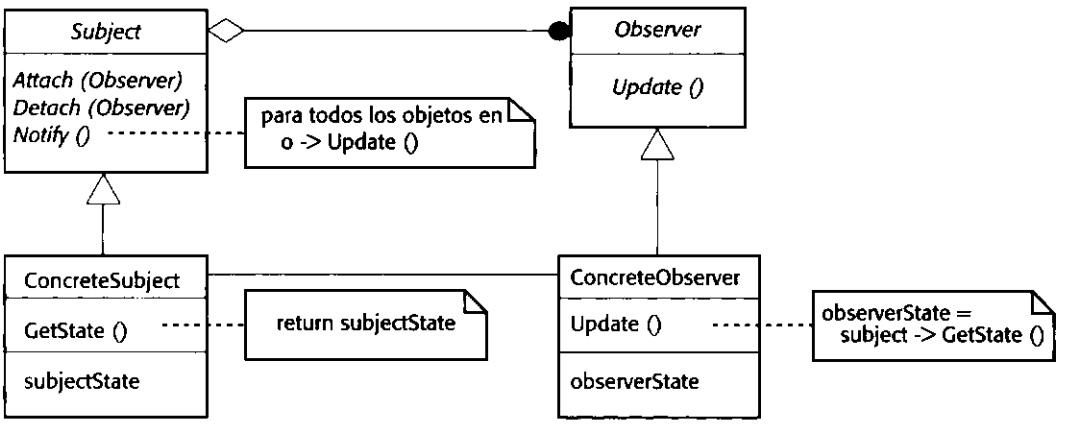


Figura 18.7 El patrón Observer.

Actualmente están disponibles un elevado número de patrones publicados (véanse las páginas web del libro para enlaces) que abarcan varios dominios de aplicaciones y lenguajes. La noción de un patrón como un concepto reutilizable ha sido desarrollada en varias áreas además del diseño software, que incluye gestión de configuraciones, diseño de interfaces de usuario y escenarios de interacciones (Berczuk y Appleton, 2002; Borchers, 2001; Martin *et al.*, 2001; Martin *et al.*, 2002).

El uso de patrones es una forma efectiva de reutilización. Sin embargo, se puede afirmar que sólo ingenieros software experimentados que tengan un conocimiento profundo de patrones pueden utilizarlos de forma efectiva. Estos desarrolladores pueden reconocer situaciones genéricas en las que se puede aplicar un patrón. Los programadores sin experiencia, aun cuando hayan leído libros sobre patrones, siempre encontrarán difícil decidir si pueden reutilizar un patrón o si necesitan desarrollar una solución de propósito específico.

18.3 Reutilización basada en generadores

El concepto de reutilización mediante patrones tiene que ver con la descripción del concepto de una forma abstracta y dejando al desarrollador la labor de crear una implementación. Una aproximación alternativa a ésta es la reutilización basada en generadores (Biggerstaff, 1998). En esta aproximación, el conocimiento reutilizable se captura en un sistema generador de programas que puede ser programado por expertos en el dominio utilizando un lenguaje orientado a dominios o una herramienta CASE interactiva que soporte la generación de sistemas. La descripción de la aplicación especifica, de forma abstracta, qué componentes reutilizables tienen que usarse y cómo tienen que ser combinados y parametrizados. Utilizando esta información se puede generar un sistema software operativo (Figura 18.8).

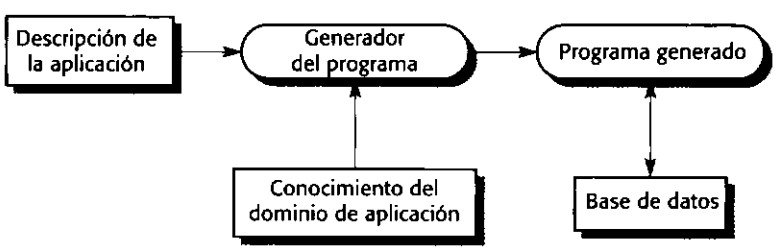


Figura 18.8
Reutilización basada en generadores.

La reutilización basada en generadores se aprovecha del hecho de que las aplicaciones del mismo dominio, tales como sistemas de negocio, tienen arquitecturas comunes y realizan funciones comparables. Por ejemplo, como se indica en el Capítulo 13, los sistemas de procesamiento de datos normalmente siguen un modelo de entrada-proceso-salida y suelen incluir operaciones tales como verificación de datos y generación de informes. Por lo tanto, pueden crearse componentes genéricos y ser incorporados en un generador de aplicaciones para seleccionar elementos de una base de datos, comprobar que éstos están dentro del rango permitido y para elaborar informes. Para reutilizar estos componentes, el programador simplemente tiene que seleccionar los datos a utilizar, las comprobaciones que deben aplicarse y el formato de los informes.

La reutilización basada en generadores ha tenido éxito sobre todo para sistemas de aplicaciones de negocios, y existen muchos productos generadores de aplicaciones de negocios disponibles. Éstos pueden generar aplicaciones completas o pueden automatizar parcialmente la creación de aplicaciones y dejar que el programador las complete con detalles específicos. La aproximación a la reutilización basada en generadores también se utiliza en otras áreas, incluyendo:

1. *Generadores de analizadores para el procesamiento del lenguaje*. La entrada del generador es una gramática que describe el lenguaje que va a ser analizado, y la salida es el analizador del lenguaje. Esta aproximación se incluye en sistemas tales como lex y yacc para C y JavaCC, un compilador de Java.
2. *Generadores de código en herramientas CASE*. La entrada de estos generadores es un diseño software y la salida es un programa que implementa el sistema diseñado. Pueden basarse en modelos UML y, dependiendo de la información en los modelos UML, generar un programa completo o componente, o bien un esqueleto de código. El desarrollador del software a continuación añade detalles para completar el código.

Estas aproximaciones a la reutilización basada en generadores se aprovechan de la estructura común de las aplicaciones en estas áreas. La técnica también ha sido usada en dominios de aplicaciones más específicos, tales como sistemas de control y de órdenes (O'Connor *et al.*, 1994) e instrumentación científica (Butler, 1994), donde se han desarrollado librerías de componentes. Los expertos del dominio utilizan un lenguaje específico del dominio para componer estos componentes y crear aplicaciones. Sin embargo, existe un alto coste inicial en la definición e implementación de los conceptos del dominio y en el lenguaje de composición. Esto significa que muchas compañías son reacias a asumir los riesgos de adoptar esta aproximación.

La reutilización basada en generadores es rentable para aplicaciones tales como procesamiento de datos de negocios. Es mucho más sencillo para los usuarios finales desarrollar programas utilizando generadores frente a otras aproximaciones basadas en componentes para la reutilización. Sin embargo, inevitablemente hay deficiencias en los programas generados. Esto significa que puede ser imposible utilizar esta aproximación en sistemas con requerimientos de elevado rendimiento.

La programación generativa es un componente clave de técnicas emergentes de desarrollo de software que combina la generación de programas con el desarrollo basado en componentes. El libro de Czarnecki y Eisenecher (Czarnecki y Eisenecher, 2000) describe estas nuevas aproximaciones.

La más desarrollada de estas aproximaciones es el desarrollo de software orientado a aspectos (AOSD) (Elrad *et al.*, 2001). El desarrollo de software orientado a aspectos aborda uno de los mayores problemas en el diseño del software: el problema de la separación de intereses. La separación de intereses es un principio de diseño básico; debería diseñarse el software

para que cada unidad o componente haga una y sólo una cosa. Por ejemplo, en el sistema LIBSYS, debería haber un componente dedicado a la búsqueda de documentos, un componente dedicado a la impresión de documentos, un componente dedicado a la gestión de descargas, y así sucesivamente.

Sin embargo, en muchas situaciones los intereses no se asocian a funciones de aplicaciones claramente definidas, sino que están compartidos; es decir, afectan a todos los componentes del sistema. Por ejemplo, supongamos que se quiere seguir la pista del uso de cada uno de los módulos del sistema por cada usuario del sistema. Por lo tanto, se tiene un interés en monitorizar que tiene que asociarse a todos los componentes. Esto no puede ser simplemente implementado como un objeto referenciado por dichos componentes. La monitorización específica que tiene que llevarse a cabo necesita información de contexto de la función del sistema que está siendo monitorizada.

En la programación orientada a aspectos, los intereses compartidos se implementan como aspectos y, dentro del programa, se define dónde se debería asociar un aspecto. Éstos se denominan *puntos de enlace*. Los aspectos se desarrollan de forma separada; a continuación, en un paso de precompilación denominado *entrelazado de aspectos*, son enlazados mediante los puntos de enlace (Figura 18.9). El entrelazado de aspectos es una forma de generación de programas; la salida del proceso de entrelazado es un programa en el que se ha integrado el código del aspecto. Un desarrollo de Java denominado AspectJ (Kiczales *et al.*, 2001) es el lenguaje mejor conocido para el desarrollo orientado a aspectos.

Actualmente el desarrollo AOSD es un tema de investigación importante, pero todavía no se ha generalizado su uso para el desarrollo industrial de software. Existen problemas con esta aproximación: el código generado nunca es tan eficiente como el código escrito manualmente, y necesitamos un mejor conocimiento de las relaciones entre los aspectos y las propiedades no funcionales del sistema. Sin embargo, en pocos años, se espera que AOSD se convierta en una aproximación importante para la reutilización del software, en donde los aspectos puedan ser reutilizados entre las aplicaciones.

18.4 Marcos de trabajo de aplicaciones

Las primeras personas que propusieron el desarrollo orientado a objetos sugirieron que los objetos eran la abstracción más adecuada para la reutilización. Sin embargo, la experiencia ha demostrado que los objetos son a menudo de grano muy fino y demasiado especializados para una aplicación particular. Por otro lado, está claro que la reutilización orientada a objetos está mejor soportada en un proceso de desarrollo orientado a objetos a través de abstracciones de grano mayor denominadas *marcos de trabajo*.

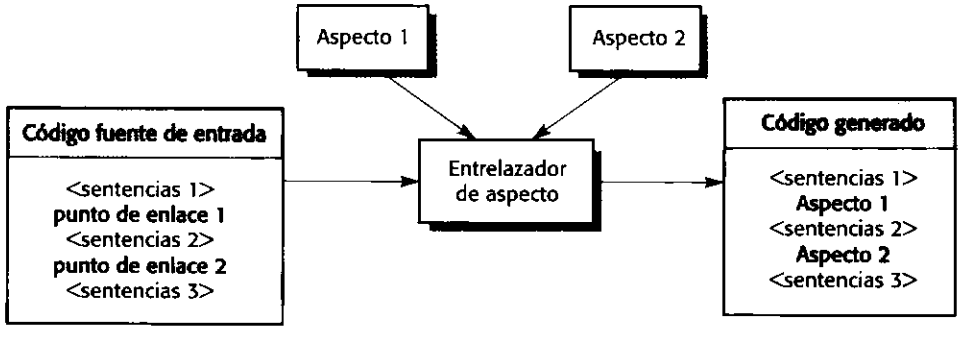


Figura 18.9 Entrelazado de aspectos.

Un marco de trabajo (o marco de trabajo de aplicaciones) es un diseño de un subsistem formado por una colección de clases concretas y abstractas y la interfaz entre ellas (Wirfs Brock y Johnson, 1990). Los detalles particulares del subsistema de aplicación son imple mentados añadiendo componentes y proporcionando implementaciones concretas de la clases abstractas en el marco de trabajo. Los marcos de trabajo raramente son aplicacione por sí mismos. Las aplicaciones se construyen normalmente integrando varios marcos d trabajo.

Fayad y Schmidt (Fayad y Schmidt, 1997) describen tres clases de marcos de trabajo:

1. *Marcos de trabajo de infraestructura de sistemas.* Estos marcos de trabajo soportan e desarrollo de infraestructuras de sistemas tales como comunicaciones, interfaces d usuario y compiladores (Schmidt, 1997).
2. *Marcos de trabajo para la integración de middleware.* Consisten en un conjunto d estándares y clases de objetos asociados que soportan la comunicación de compc nentes y el intercambio de información. Ejemplos de este tipo de marcos son COR BA, COM+ de Microsoft y Enterprise Java Beans. Estos marcos proporcionan sc porte para modelos de componentes estandarizados, tal y como se expone en e Capítulo 19.
3. *Marcos de trabajo de aplicaciones empresariales.* Se refieren a dominios de aplic ciones específicos tales como telecomunicaciones o sistemas financieros (Baumer *al.*, 1997). Estos marcos de trabajo encapsulan el conocimiento del dominio de la apl cación y soportan el desarrollo de aplicaciones para los usuarios finales.

Tal y como el nombre sugiere, un marco de trabajo es una estructura genérica que pued ser extendida para crear un subsistema o aplicación más específico. Éste es implementad como una colección de clases de objetos concretas y abstractas. Para extender el marco de tr bajo, se tienen que añadir clases concretas que hereden operaciones de las clases abstract en el marco. Además se deben definir *callbacks*. Los callbacks son métodos que se llama como respuesta a eventos reconocidos por el marco de trabajo.

Uno de los marcos de trabajo más conocido y ampliamente usado para el diseño de GU es el marco Modelo-Vista-Controlador (MVC) (Figura 18.10). El marco de trabajo MVC fu propuesto originalmente en la década de los 80 como una aproximación al diseño de GU que permitió múltiples presentaciones de un objeto y estilos independientes de interacción cc cada una de estas presentaciones. El marco MVC soporta la presentación de los datos de d ferentes formas (véase la Figura 18.6) e interacciones independientes con cada una de est presentaciones. Cuando los datos se modifican a través de una de las presentaciones, el res de las presentaciones son actualizadas.

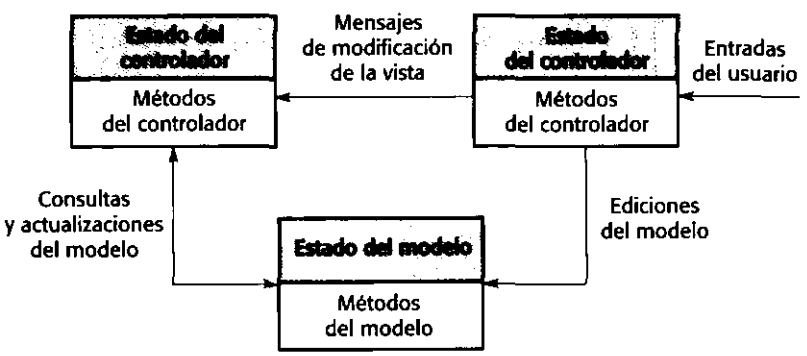


Figura 18.10
El marco de trabajo Modelo-Vista-Controlador.

Los marcos de trabajo son a menudo instanciaciones de varios patrones, tal y como se vio en la Sección 18.2. Por ejemplo, el marco MVC incluye el patrón Observer descrito en la Figura 18.5, el patrón Strategy relacionado con la actualización del modelo, el patrón Composite y otros patrones descritos por Gamma y colaboradores (Gamma *et al.*, 1995).

Las aplicaciones construidas utilizando marcos de trabajo pueden ser las bases para una posterior reutilización a través del concepto de líneas de productos software o familias de aplicaciones, tal y como se indica en la Sección 18.5.2. Debido a que estas aplicaciones se construyen utilizando un marco, se simplifica la modificación de miembros de la familia para crear nuevos miembros.

El problema fundamental con los marcos de trabajo es su complejidad inherente y el tiempo que lleva aprender a utilizarlos. Pueden requerirse varios meses para comprender completamente el marco de trabajo, por lo que es muy probable que, en organizaciones grandes, algunos ingenieros software se conviertan en especialistas en marcos de trabajo. No hay duda de que ésta es una aproximación efectiva para la reutilización, pero es muy elevado el coste que supone introducirla en los procesos de desarrollo del software.

18.5 Reutilización de sistemas de aplicaciones

La reutilización de sistemas de aplicaciones implica reutilizar sistemas de aplicaciones completos configurando un sistema para un entorno específico o integrando dos o más sistemas para crear una nueva aplicación. Tal y como se sugirió en la Sección 18.1, la reutilización de sistemas de aplicaciones es a menudo la técnica de reutilización más efectiva. Implica la reutilización de activos de grano grueso que pueden ser rápidamente configurados para crear un nuevo sistema.

En esta sección, se analizan dos tipos de reutilización de aplicaciones: la creación de nuevos sistemas integrando dos o más aplicaciones comerciales y el desarrollo de líneas de productos. Una línea de productos es un conjunto de sistemas basados en una arquitectura base común y componentes compartidos. El sistema base se diseña de forma específica para ser configurado y adaptado a fin de adecuarse a las necesidades específicas de los diferentes clientes del sistema.

18.5.1 Reutilización de productos COTS

La denominación producto COTS se aplica a un sistema software que puede utilizarse sin cambios por su comprador. Virtualmente todo el software de sobremesa y un gran número de productos servidores son software COTS. Debido a que este software se diseña para uso general, normalmente incluye muchas características y funciones para que sea potencialmente reutilizable en diferentes aplicaciones y entornos. Si bien puede haber problemas con esta aproximación para la construcción de sistemas (Tracz, 2001), existe un número creciente de experiencias con éxito que demuestran su viabilidad (Baker, 2002; Balk y Kedia, 2000; Pfarr y Reis, 2002).

Algunos tipos de productos COTS han sido reutilizados durante muchos años. Los sistemas de bases de datos constituyen quizá el mejor ejemplo de esto. Muy pocos desarrolladores podrían tomar en consideración la implementación de su propio sistema de gestión de base de datos. Sin embargo, hasta mediados de los 90, solamente existían unos cuantos sistemas grandes, como los sistemas de gestión de bases de datos y monitores de teleprocesamiento,

que fueran reutilizados de forma rutinaria. La mayoría de los sistemas grandes se diseñab como sistemas únicos, y a menudo había muchos problemas en conseguir que estos sistem trabajasen de forma conjunta.

En la actualidad, es normal para los sistemas grandes el tener definidas Interfaces Programación de Aplicaciones (APIs) que permiten programar el acceso a las funciones dichos sistemas. Esto significa que la creación de grandes sistemas tales como sistemas de c mercio electrónico mediante la integración de varios sistemas COTS debería considerar siempre como una opción seria de diseño. Debido a la funcionalidad que estos product COTS ofrecen, es posible reducir costes y tiempos de entrega en varios órdenes de magnitu comparados con el desarrollo de nuevo software. Además, los riesgos pueden reducirse ya q el producto se encuentra disponible y los gestores pueden ver si dicho producto satisface s requerimientos.

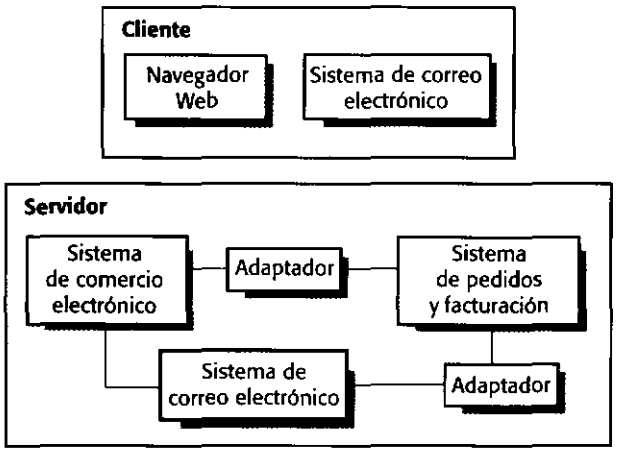
Para desarrollar sistemas utilizando productos COTS, se tienen que tomar varias elecci nes de diseño:

1. *¿Qué productos COTS ofrecen la funcionalidad más adecuada?* Si todavía no se ti ne experiencia con un producto COTS, puede ser difícil decidir qué producto es el m adecuado.
2. *¿Cómo se intercambiarán los datos?* En general, los productos individuales utiliz estructuras de datos y formatos únicos, y se tienen que escribir adaptadores para co vertir una representación en otra.
3. *¿Qué características de un producto se utilizarán realmente?* La mayoría de los pr ductos COTS poseen más funcionalidades de las que se necesitan, y las funcionalid des a menudo se duplican entre diferentes productos. Se tiene que decidir q características de qué productos son las más adecuadas para los propios requerimie tos. Si es posible, debería también denegarse el acceso a funcionalidades no utilizad debido a que pueden interferir en el funcionamiento normal del sistema. El fallo del p mer vuelo del cohete Ariane 5, comentado en el Capítulo 19 (Nuseibeh, 1997), se d bió a un fallo en una funcionalidad no utilizada de un subsistema reutilizado.

Como ejemplo de una integración COTS, supongamos que una gran organización quie desarrollar un sistema de adquisiciones que permite al personal emitir órdenes desde su equ po de sobremesa. Introduciendo este sistema dentro de la organización, la compañía estim que puede ahorrar 5 millones de dólares al año. Al centralizar las compras, el nuevo siste de adquisiciones puede asegurar que los pedidos siempre se realizan a los proveedores q ofertan los mejores precios y se deberían reducir los costes del papeleo asociado con los p didos. Al igual que ocurre con los sistemas manuales, esto implica elegir los productos di ponibles de un proveedor, crear un pedido, obtener la aprobación del pedido, enviar el ped do a un proveedor, recibir los productos y confirmar que los pagos se efectúen.

La empresa ya posee un sistema de pedidos que es utilizado por el departamento de a quisiciones. Éste ya está integrado con sus sistemas de facturación y reparto. Para crear el nu vo sistema de pedidos, se integra el sistema antiguo con una plataforma de comercio electr nico basada en web y un sistema de correo electrónico que gestiona las comunicaciones c los usuarios. La estructura del sistema de adquisiciones resultante construido utilizando COT se muestra en la Figura 18.11.

Este sistema de adquisiciones es un sistema cliente-servidor, y en el cliente se utiliza u navegador web y un software de correo electrónico estándares. Éstos ya están integrados p los proveedores del software. En la parte del servidor, la plataforma de comercio electróni tiene que integrarse con el sistema de adquisiciones existente mediante un adaptador. El si



gura 18.11
n sistema de
lquisiciones
sado en COTS.

tema de comercio electrónico tiene su propio formato para los pedidos, conformidades con las entregas, y así sucesivamente, los cuales tienen que convertirse al formato utilizado por el sistema de pedidos. En el sistema de comercio electrónico está integrado el sistema de correo electrónico para enviar las notificaciones a los usuarios, pero el sistema de pedidos nunca fue diseñado para esto. Por lo tanto, tiene que desarrollarse otro adaptador para convertir las notificaciones en mensajes de correo electrónico.

En principio, el uso de un sistema COTS a gran escala es el mismo que el uso de cualquier otro componente más específico. Se tienen que comprender las interfaces del sistema y hay que utilizarlas exclusivamente para comunicarse con el componente; se tiene que buscar un equilibrio entre los requerimientos específicos y el desarrollo rápido y la reutilización; y se tiene que diseñar una arquitectura del sistema que permita que los sistemas COTS operen conjuntamente.

Sin embargo, el hecho de que estos productos son normalmente sistemas grandes y a menudo se venden como sistemas independientes y únicos, introduce problemas adicionales. Boehm y Abts (Boehm y Abts, 1999) plantean cuatro problemas con la integración de sistemas COTS:

1. *Ausencia de control sobre la funcionalidad y el rendimiento.* Si bien la interfaz que presenta un producto puede parecer que oferte las facilidades requeridas, éstas pueden no estar implementadas adecuadamente o pueden tener un pobre rendimiento. El producto puede tener operaciones ocultas que interfieran en su funcionamiento en una situación específica. Resolver estos problemas debe ser una prioridad para el integrador del producto COTS, pero no es un problema que concierna al vendedor del producto. Los usuarios simplemente tienen que buscar la forma de sortear los problemas si quieren reutilizar el producto COTS.
2. *Problemas con la interoperabilidad del sistema COTS.* Algunas veces es difícil obtener productos COTS que trabajen de forma conjunta debido a que cada producto se basa en sus propias suposiciones de cómo debería utilizarse. Garlan y otros (Garlan *et al.*, 1995) publicaron su experiencia en el intento de integración de cuatro productos COTS, y observaron que tres de estos productos estaban basados en eventos, pero que cada uno usaba un modelo diferente de eventos, y consideraron que cada uno de dichos modelos tenía acceso exclusivo a la cola de eventos. Como consecuencia, el proyecto requirió cinco veces más esfuerzo que el que se predijo originalmente, y la agenda se prolongó durante dos años en lugar de los seis meses previstos.

3. *No existe control sobre la evolución del sistema.* Los vendedores de productos COT toman sus propias decisiones sobre los cambios en el sistema como respuesta a las pr siones del mercado. Concretamente, en productos para PCs se producen frecuent mente nuevas versiones y pueden no ser compatibles con todas las versiones anteri res. Las nuevas versiones pueden tener funcionalidades adicionales no deseadas, versiones previas pueden dejar de estar disponibles y no tener soporte.
4. *Soporte de los vendedores de productos COTS.* El nivel de soporte disponible desc los vendedores de COTS es importante cuando surgen problemas debido a que l desarrolladores no tienen acceso al código fuente y a la documentación detallada d sistema. Si bien los vendedores pueden comprometerse a proporcionar soporte, l cambios en el mercado y las circunstancias económicas pueden hacer que sea difíc para ellos proporcionar dicho soporte. Por ejemplo, un vendedor de sistemas COT puede decidir no continuar vendiendo un producto debido a una escasa demanda puede ser absorbido por otra compañía que no desea soportar todos sus productos a tuales.

Desde luego, no es probable que todos estos problemas ocurran en cada caso, pero al m nos uno de ellos se va a producir en la mayoría de los proyectos de integración de COTS. E consecuencia, los beneficios en coste y tiempo derivados de la reutilización de COTS son pr bablemente menores que los que podrían esperarse en un principio.

Además, Boehm y Abts señalan que, en muchos casos, el coste de mantenimiento y ev lución de un sistema puede ser mayor cuando se utilizan productos COTS. Todas las dificu tades anteriores son problemas del ciclo de vida; no sólo afectan al desarrollo inicial del si tema. En un sistema, cuantos menos desarrolladores originales haya, más personas estar implicadas en su mantenimiento, por lo que es más probable que se planteen problemas c la integración de productos COTS.

A pesar de estos problemas, los beneficios de la reutilización de productos COTS son si nificativos debido a que estos sistemas ofrecen mucha más funcionalidad a la persona q los reutiliza. Pueden ahorrarse meses y a veces años de esfuerzo de implementación si se re tiliza un sistema existente y los tiempos de desarrollo del sistema se reducen drásticamen Por ejemplo, el sistema de adquisiciones descrito en la Figura 18.11 fue implementado y d plegado en una compañía muy grande en nueve meses, mientras que originalmente se es maron tres años para el desarrollo de un sistema nuevo. Si la entrega de un sistema de f ma rápida es fundamental y se tiene alguna flexibilidad en los requerimientos, entonces integración de productos COTS es a menudo la estrategia de reutilización más efectiva a s guir.

18.5.2 Líneas de productos software

Una de las aproximaciones más efectivas para la reutilización es la creación de líneas de p ductos software o familias de aplicaciones. Una línea de productos es un conjunto de apli ciones con una arquitectura común específica de dichas aplicaciones, tal y como se expuso el Capítulo 13. Cada aplicación específica se especializa de alguna manera. El núcleo com de la familia de aplicaciones se reutiliza cada vez que se requiere una nueva aplicación. El n vo desarrollo puede implicar una configuración específica de componentes, implementaci de componentes adicionales y adaptación de algunos componentes para satisfacer las nue demandas.

Se pueden desarrollar varios tipos de especialización de líneas de productos software:

1. *Especialización de la plataforma.* Se desarrollan versiones de la aplicación para diferentes plataformas. Por ejemplo, pueden existir versiones de la aplicación para plataformas Windows, Solaris y Linux. En este caso, la funcionalidad de la aplicación normalmente no se cambia; sólo se modifican aquellos componentes que interactúan con el hardware y con el sistema operativo.
2. *Especialización del entorno.* Se crean versiones de la aplicación para gestionar entornos operativos y dispositivos periféricos concretos. Por ejemplo, un sistema para servicios de emergencia puede existir en distintas versiones dependiendo del tipo de sistema de radio utilizado. En este caso, los componentes del sistema se cambian para reflejar la funcionalidad del equipo de comunicaciones utilizado.
3. *Especialización de la funcionalidad.* Se crean versiones de la aplicación para clientes específicos que tienen diferentes requerimientos. Por ejemplo, un sistema automático de biblioteca puede modificarse dependiendo de si es utilizado en una biblioteca pública, una biblioteca privada o una biblioteca universitaria. En este caso, los componentes que implementan la funcionalidad pueden ser modificados y se pueden añadir nuevos componentes al sistema.
4. *Especialización del proceso.* El sistema se adapta para tratar con procesos de negocio específicos. Por ejemplo, un sistema de pedidos puede adaptarse en una compañía para trabajar con un proceso de pedidos centralizado y con un proceso distribuido en otra.

Las líneas de productos software se diseñan para ser reconfiguradas. Esta reconfiguración puede implicar añadir o eliminar componentes del sistema, definir parámetros y restricciones para los componentes del sistema, e incluir conocimiento de los procesos de negocio. Las líneas de productos software pueden ser configuradas en dos puntos del proceso de desarrollo:

- *Configuración durante el despliegue*, en donde un sistema genérico se diseña para su configuración por un cliente o consultores que trabajan con el cliente. El conocimiento de los requerimientos específicos del cliente y el entorno del sistema operativo se incluye en un conjunto de ficheros de configuración que son utilizados por el sistema genérico.
- *Configuración durante el diseño*, en donde la organización que está desarrollando el software modifica un núcleo de líneas de productos comunes desarrollando, seleccionando o adaptando componentes para crear un nuevo sistema para un cliente.

La configuración durante el despliegue es la aproximación utilizada en paquetes de software verticales que son diseñados para una aplicación específica tal como un sistema de gestión de información de un hospital. También se utiliza en sistemas de Planificación de Recursos de Empresas (ERP) (O'Leary, 2000) tales como los producidos por SAP y BEA. Éstos son sistemas integrados y a gran escala, que son diseñados para soportar procesos de negocio tales como pedidos y facturación, gestión de inventarios y planificación de fabricación. El proceso de configuración para estos sistemas implica compartir la información detallada sobre el negocio del cliente y el proceso de negocio y a continuación incluir esta información en una base de datos de configuraciones. Esto a menudo requiere el conocimiento detallado de las herramientas y notaciones de configuración y normalmente se lleva a cabo por consultores que trabajan al lado de los clientes. La Figura 18.12 ilustra la organización de un sistema ERP.

El sistema ERP genérico incluye un gran número de módulos que pueden componerse de diferentes formas para crear un sistema específico. El proceso de configuración implica elegir qué módulos tienen que ser incluidos, configurar estos módulos individuales, definir procesos y reglas de negocio, y definir la estructura y organización de la base de datos del sistema.

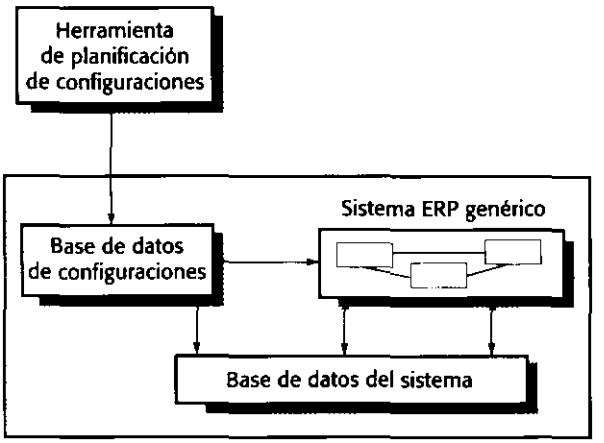


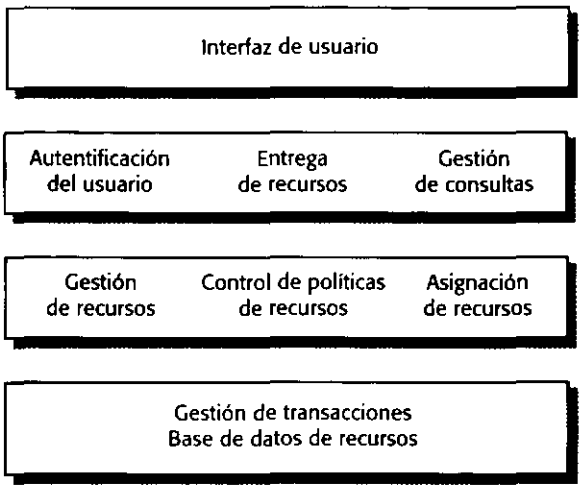
Figura 18.12 Configuración de un sistema ERP.

Los sistemas ERP son quizá el ejemplo más extendido de reutilización de software. La m yoría de las grandes compañías utilizan estos sistemas para soportar alguna o todas sus fu ciones. Sin embargo, existe la limitación obvia de que la funcionalidad del sistema se restrin a la funcionalidad del núcleo genérico. Además, las operaciones y procesos de una compañ tienen que expresarse en el lenguaje de configuración del sistema, y puede haber una discr pancia entre los conceptos del negocio y los conceptos soportados en el lenguaje de config ración. Por ejemplo, en un sistema ERP que fue vendido a una universidad, el concepto de cliente tuvo que ser definido. Esto provocó problemas reales debido a que las universidad tienen múltiples tipos de clientes (estudiantes, agencias de investigación, instituciones ed cativas, etc.) y ninguno de éstos es comparable con un cliente comercial. Una seria discr pancia entre el modelo de negocio utilizado por el sistema y el modelo utilizado por el clie te hace que sea muy probable que el sistema ERP no satisfaga las necesidades reales d cliente (Scott, 1999).

La aproximación alternativa a la reutilización de familias de aplicaciones es la configur ción por el proveedor del sistema antes de entregarlo al cliente. El proveedor comienza c un sistema genérico y a continuación, modificando y extendiendo módulos en este siste crea un sistema específico que proporciona las funcionalidades requeridas por el cliente. Es aproximación normalmente implica cambiar y extender el código fuente del núcleo del sis ma para que sea posible una mayor flexibilidad que con una configuración durante el de pliegue.

Las líneas de productos software normalmente surgen a partir de aplicaciones existent Es decir, una organización desarrolla una aplicación y, cuando se requiere una nueva aplic ción, se utiliza la primera como base para la nueva aplicación. Posteriores demandas de nu vas aplicaciones hacen que el proceso continúe. Sin embargo, debido a que los cambios tie den a degradar la estructura de la aplicación, en algún momento debe tomarse alguna decisi específica para diseñar una línea de productos genérica. El diseño se basa en la reutilizaci del conocimiento adquirido a partir del desarrollo del conjunto inicial de aplicaciones.

Se puede pensar en las líneas de productos software como instanciaciones y especializ ciones de arquitecturas de aplicaciones más genéricas, tal y como se explicó en el Capítu 13. Una arquitectura de aplicaciones es muy general; las líneas de productos software esp cializan la arquitectura para un tipo concreto de aplicación. Por ejemplo, consideremos un s tema de líneas de productos diseñado para gestionar el uso de un vehículo para servicios emergencia. Los operadores de este sistema recogen llamadas sobre incidentes ocurridos, e

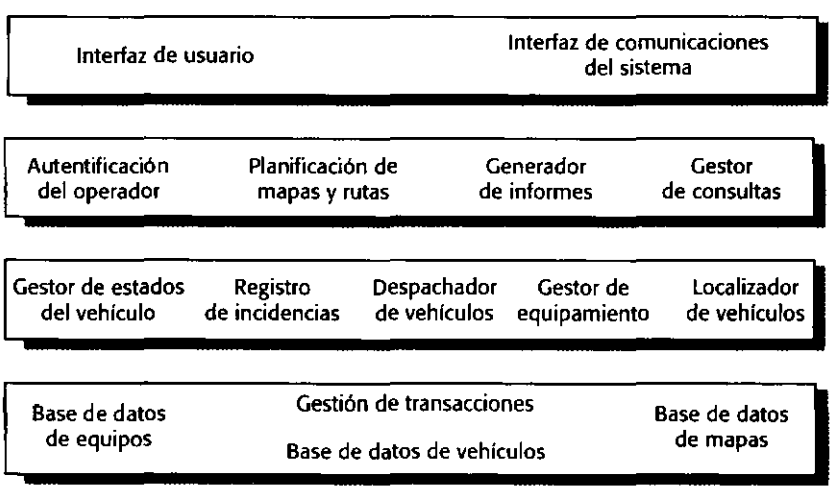


gura 18.13
rquitectura de un
stema de
signación de
cursos.

cuentran el vehículo adecuado para responder al incidente y envían el vehículo al lugar del incidente. Los desarrolladores de un sistema como éste pueden comercializar versiones de dicho sistema para servicios de policía, de bomberos y de ambulancia.

Este sistema de gestión de vehículos es un ejemplo de un sistema de gestión de recursos cuya arquitectura de las aplicaciones se muestra en la Figura 18.13. Puede verse cómo se instancia esta estructura de cuatro capas en la Figura 18.14, que muestra los módulos que podrían incluirse en una línea de productos de sistemas de gestión de vehículos. Los componentes en cada nivel de la línea de productos son los siguientes:

1. En el nivel de interfaz de usuario, hay componentes que proporcionan una interfaz de la pantalla del operador y una interfaz con el sistema de comunicaciones utilizado.
2. En el nivel de E/S (nivel 2), hay componentes que gestionan la autenticación del operador, generan informes de incidentes y vehículos enviados, soportan la generación de mapas y planificación de rutas, y proporcionan un mecanismo para los operadores para realizar consultas a las bases de datos del sistema.



gura 18.14
rquitectura de línea
e productos de un
stema de gestión
vehículos.

3. En el nivel de gestión de recursos (nivel 3), hay componentes que permiten localiz y enviar los vehículos al lugar del incidente, componentes que actualizan el estado los vehículos y del equipamiento, y un componente para registrar los detalles de l incidentes.
4. En el nivel de base de datos, además del soporte usual para la gestión de transacci nes, hay bases de datos independientes de vehículos, equipamiento y mapas.

Para crear una versión específica de este sistema, se puede tener que modificar comp nentes individuales. Por ejemplo, la policía tiene un gran número de vehículos pero pocos t pos de vehículos, mientras que el servicio de incendios tiene muchos tipos de vehículos e pecializados; por lo tanto, puede ser necesario incorporar al sistema una estructura diferen de base de datos de vehículos.

La Figura 18.15 muestra los pasos comprendidos en la adaptación de una familia aplicaciones para crear una nueva aplicación. Los pasos implicados en este proceso g neral son:

1. *Elicitación de los requerimientos de los stakeholders*. Se puede empezar con un pr ceso normal de ingeniería de requerimientos. Sin embargo, debido a que un siste ya existe, se necesitará probar y experimentar con ese sistema, expresando sus requ rimientos como modificaciones para las funciones ya proporcionadas.
2. *Elegir un miembro adecuado de la familia*. Se analizan los requerimientos y se eli el miembro de la familia que más se adecue a los requerimientos para su modificació Éste no necesita ser el sistema que fue probado.
3. *Renegociar los requerimientos*. A medida que surgen más detalles de cambios requ ridos y se planifica el proyecto, puede haber alguna renegociación de requerimient para minimizar los cambios que se necesitan.
4. *Adaptar el sistema existente*. Se desarrollan nuevos módulos para el sistema existe te, y los módulos del sistema existente se adaptan para satisfacer los nuevos requer mientos.
5. *Entregar el nuevo miembro de la familia*. La nueva instancia de la línea de product se entrega al cliente. En esta etapa, se deberían documentar sus características cla para que pueda utilizarse como base para otros desarrollos futuros de sistemas.

Cuando se crea un nuevo miembro de una familia de aplicaciones, se puede tener que bu car un equilibrio entre reutilizar la aplicación genérica tanto como sea posible y satisfacer l requerimientos detallados de los *stakeholders*. Cuanto más detallados sean los requerimie tos del sistema, es menos probable que existan componentes que satisfagan dichos requer mientos. Sin embargo, si los *stakeholders* están dispuestos a ser flexibles y a limitar las m dificaciones requeridas del sistema, normalmente se entregará el sistema más rápidamente con un menor coste.

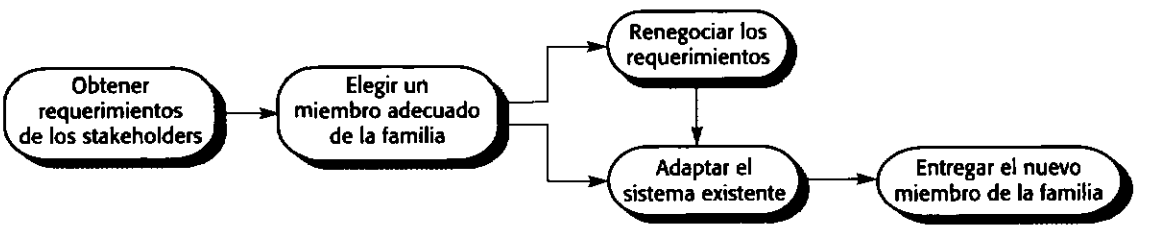


Figura 18.15 Desarrollo de una instancia de un producto.

En general, el desarrollo de aplicaciones adaptando una versión genérica de la aplicación significa que una proporción muy elevada del código de la aplicación es reutilizada. Además, la experiencia de aplicaciones es a menudo transferible de un sistema a otro, de forma que cuando los ingenieros software se incorporan a un equipo de desarrollo, su proceso de aprendizaje se reduce. Las pruebas se simplifican debido a que las pruebas para partes grandes de la aplicación también pueden ser reutilizadas, reduciendo así el tiempo total del desarrollo de la aplicación.

PUNTOS CLAVE

- **Las ventajas de la reutilización del software son costes más bajos, desarrollo del software más rápido y menores riesgos. La confiabilidad del sistema se incrementa y los especialistas pueden ser utilizados de forma más efectiva concentrando su experiencia en el diseño de componentes reutilizables.**
- **Los patrones de diseño son abstracciones de alto nivel que documentan soluciones de diseño exitosas. Son fundamentales para reutilizar el diseño en el desarrollo orientado a objetos. Una descripción del patrón debería incluir un nombre del patrón, una descripción del problema y la solución, y una declaración de los resultados y compromisos al utilizar el patrón.**
- **Los generadores de programas son una aproximación alternativa al concepto de reutilización en donde los conceptos reutilizables están embebidos en un sistema generador. El diseñador especifica las abstracciones requeridas utilizando un lenguaje específico del dominio, y se genera un programa ejecutable.**
- **Los marcos de trabajo son colecciones de objetos concretos y abstractos que se diseñan para ser reutilizados mediante la especialización y la adición de nuevos objetos.**
- **La reutilización de productos COTS está relacionada con la reutilización a gran escala de los sistemas comerciales. Éstos proporcionan bastante funcionalidad, y su reutilización puede reducir los costes y tiempo de desarrollo de forma radical.**
- **Los problemas potenciales con la reutilización basada en COTS incluyen la ausencia de control sobre la funcionalidad y el rendimiento, ausencia de control sobre la evolución de los sistemas, la necesidad de soporte de vendedores externos y dificultades para asegurar que los sistemas pueden interoperar.**
- **Los sistemas de Planificación de Recursos de Empresa son usados de forma muy extensa. Se crean sistemas ERP específicos configurando un sistema genérico en tiempo de despliegue con información sobre el negocio del cliente.**
- **Las líneas de productos software son aplicaciones relacionadas que se desarrollan a partir de una o más aplicaciones base. Un sistema genérico se adapta y especializa para satisfacer requerimientos específicos de funcionalidades, plataforma de destino o configuración de funcionamiento.**

LECTURAS ADICIONALES

Reuse-based Software Engineering. Un estudio claro de diferentes aproximaciones para la reutilización del software. Los autores tratan cuestiones técnicas de la reutilización y procesos de gestión de reutilización. (H. Mili *et al.*, 2002, John Wiley & Sons.)

«A Lifecycle Process for the effective reuse of commercial off-the-shelf software». Ésta es una buena introducció general, que estudia las ventajas y desventajas de utilizar COTS en ingeniería del software. (C. L. Braun, *Pro Symposium on Software Reusability*, Los Ángeles, 1999. ACM Press. Disponible desde la Librería Digital ACM.)

Design Patterns: Elements of Reusable Object-oriented Software. Ésta es la guía de referencia original de patrone software que introdujo este concepto a una comunidad amplia. (E. Gamma *et al.*, 1995, Addison-Wesley.)

«Aspect-oriented programming». Este número especial de la CACM tiene varios artículos sobre el desarrollo de sof ware orientado a aspectos. Es un excelente punto de partida para leer sobre este tema. [*Comm. ACM*, 44(10), octu bre de 2001.]

EJERCICIOS

18.1 ¿Cuáles son los principales factores técnicos y no técnicos que impiden la reutilización de software? ¿Us ted reutiliza mucho software?, y si no lo hace, ¿por qué?

18.2 Sugiera por qué los ahorros de costes al reutilizar software existente no son simplemente proporcionale al tamaño de los componentes que son reutilizados.

18.3 Dé cuatro circunstancias en que usted podría no recomendar la reutilización de software.

18.4 ¿Por qué los patrones son una forma efectiva de reutilización para el diseño? ¿Cuáles son las desventaja de esta aproximación para la reutilización?

18.5 Además de los dominios de aplicaciones discutidos aquí sugiera otros dos dominios en donde la reuti zación basada en un generador podría tener éxito. Explique por qué piensa que esta aproximación para reutilización será rentable en estos dominios.

18.6 Explique por qué se necesitan normalmente adaptadores cuando los sistemas se construyen integrand productos COTS.

18.7 Identifique seis posibles riesgos que pueden tener lugar cuando los sistemas se construyen utilizand COTS. ¿Qué pasos puede seguir una compañía para reducir estos riesgos?

18.8 Utilizando una arquitectura general de un sistema de información (descrita en el Capítulo 13) como pun de partida, diseñe una familia de aplicaciones de sistemas de información que podría utilizarse en bibli tecas de libros, películas, música y recortes de periódico.

18.9 Utilizando el ejemplo del sistema de estación meteorológica descrito en el Capítulo 14, sugiera una arqu tectura para una familia de aplicaciones que esté relacionada con la monitorización remota y la recole ción de datos.

18.10 La reutilización de software trae consigo varias cuestiones sobre derechos de autor y propiedad intele tual. Si un cliente paga a un proveedor de software para desarrollar un sistema, ¿quién tiene el derech de reutilización del código desarrollado? ¿El proveedor de software tiene derecho de utilizar ese códig como base para un componente genérico? ¿Qué mecanismos de pago podrían utilizarse para reembols a los proveedores de componentes reutilizables? Comente estas y otras cuestiones éticas asociadas a reutilización de software.

19 Ingeniería del software basada en componentes

Objetivos

El objetivo de este capítulo es describir un proceso de desarrollo del software basado en la composición de componentes reutilizables y estandarizados. Cuando haya leído este capítulo:

- conocerá que la ingeniería del software basada en componentes está relacionada con el desarrollo de componentes estandarizados basados en un modelo de componentes y la composición de éstos en sistemas de aplicaciones;
- comprenderá el concepto de componente y modelo de componentes;
- conocerá las principales actividades en el proceso CBSE y comprenderá por qué tiene que llegar a un compromiso con los requerimientos para que los componentes puedan ser reutilizados;
- comprenderá algunas de las dificultades y problemas que tienen lugar durante el proceso de composición de componentes.

Contenidos

Tal y como se sugirió en el Capítulo 18, la ingeniería del software basada en reutilización s está convirtiendo en la principal aproximación de desarrollo para sistemas comerciales y d empresas. Las entidades que son reutilizadas varían desde funciones de grano fino hasta sis temas enteros de aplicaciones. Sin embargo, hasta hace relativamente poco, ha sido difícil re utilizar componentes de programas de grano medio. Los componentes de grano medio so significativamente más grandes que los objetos individuales o procedimientos, con más fun cionalidad, aunque son más pequeños y más específicos que los sistemas de aplicación. Afor tunadamente, los desarrollos estandarizados promovidos por los principales vendedores d software han dado lugar actualmente a que los componentes puedan interoperar dentro de u marco de trabajo como CORBA. Éste ha abierto oportunidades para la reutilización sistemá tica a través de la ingeniería del software basada en componentes.

La ingeniería del software basada en componentes (CBSE) surgió a finales de los 90 com una aproximación basada en reutilización al desarrollo de sistemas software. Su creación fu motivada por la frustración de los diseñadores de que el desarrollo orientado a objetos no h conducido a una reutilización extensiva, tal y como se sugirió originalmente. Las clases de ob jetos simples eran demasiado detalladas y específicas, y a menudo tenían que ser enlazada con aplicaciones en tiempo de compilación. Había que tener un conocimiento detallado de la clases para usarlas, lo cual generalmente significaba que había que tener acceso al códig fuente del componente. Esto hizo que fuera difícil el marketing de objetos como componen tes reutilizables. A pesar de las predicciones optimistas iniciales, nunca se desarrolló un mer cado significativo para objetos individuales.

CBSE es el proceso de definir, implementar e integrar o componer en sistemas compo nentes independientes débilmente acoplados. Se ha convertido en una importante aproxima ción de desarrollo del software debido a que los sistemas software son cada vez más grande y más complejos y los clientes demandan software más confiable que sea desarrollado má rápidamente. La única forma en la que podemos tratar con la complejidad y entregar mejo software más rápidamente es reutilizar componentes software en vez de reimplementarlos.

Los fundamentos de la ingeniería del software basada en componentes son:

1. *Componentes independientes*, que son completamente especificados por sus interfa ces. Debería haber una clara separación entre la interfaz de los componentes y su im plementación para que una implementación de un componente pueda reemplazars por otro sin cambiar el sistema.
2. *Estándares de componentes*, que facilitan la integración de los componentes. Estos es tándares se incluyen en un modelo de componentes y definen, en el nivel más baj cómo las interfaces de componentes deberían especificarse y cómo se comunican lo componentes. Algunos modelos definen interfaces que deberían implementarse con forme a todos los componentes. Si los componentes cumplen con los estándares, en tonces su funcionamiento es independiente de su lenguaje de programación. Los com ponentes escritos en diferentes lenguajes pueden integrarse en el mismo sistema.
3. *El middleware*, que proporciona soportes software para la integración de componen tes. Para conseguir que componentes independientes y distribuidos trabajen juntos, s necesita un soporte middleware que maneje las comunicaciones de los componente Un producto middleware como CORBA (Pope, 1998), tratado en el Capítulo 12, m neja cuestiones de bajo nivel de forma eficiente y permite al diseñador centrarse e problemas relacionados con la aplicación. Además, un producto middleware utiliz do para implementar un modelo de componentes puede proporcionar soporte pa asignación de recursos, gestión de transacciones, seguridad y concurrencia.

4. *Un proceso de desarrollo*, que se adapta a la ingeniería del software basada en componentes. Si se intenta añadir una aproximación basada en componentes a un proceso de desarrollo que está adaptado a la producción de software original, se puede observar que las suposiciones inherentes al proceso limitan el potencial del CBSE. Los procesos de desarrollo CBSE se analizan en la Sección 19.2.

El desarrollo basado en componentes se está adoptando cada vez más como una aproximación fundamental a la ingeniería del software incluso aunque los componentes reutilizables no estén disponibles. CBSE está asentado sobre unos principios de diseño sólidos que soportan la construcción de software comprensible y mantenible. Los componentes son independientes para que no interfieran en su funcionamiento unos con otros. Los detalles de implementación se ocultan, por lo que la implementación de los componentes puede cambiarse sin afectar al resto del sistema. Los componentes se comunican a través de interfaces bien definidas, de forma que si estas interfaces se mantienen, un componente puede reemplazarse por otro que proporcione una funcionalidad adicional o mejorada. Además, las infraestructuras de componentes proporcionan plataformas de alto nivel que reducen los costes del desarrollo de aplicaciones.

Aunque CBSE se está convirtiendo en una aproximación fundamental para el desarrollo del software, presenta varios problemas:

1. *Confiabilidad de los componentes.* Los componentes son cajas negras de unidades de programas, y el código de los componentes puede no estar disponible para los usuarios de dichos componentes. En tales casos, ¿cómo sabe un usuario que un componente es confiable? El componente puede no tener documentados modos de fallo que comprometen el sistema en el que se utiliza dicho componente. Su comportamiento no funcional puede no ser el esperado, y un problema más grave es que el componente podría ser un caballo de Troya que oculta código dañino que viola la seguridad del sistema.
2. *Certificación de componentes.* Estrechamente relacionada con la confiabilidad se halla la cuestión de la certificación. Se ha propuesto que asesores independientes deberían certificar los componentes para asegurar a los usuarios que los componentes son confiables. Sin embargo, no está claro cómo se puede hacer esto. ¿Quién pagaría por la certificación?, ¿quién sería responsable si el componente no funciona de acuerdo con la certificación?, y ¿cómo podrían los certificadores limitar su responsabilidad? Desde nuestro punto de vista, la única solución viable es certificar que los componentes cumplen una especificación formal. Sin embargo, la industria no parece dispuesta a pagar por esto.
3. *Predicción de propiedades emergentes.* Tal y como se explicó en el Capítulo 2, todos los sistemas tienen propiedades emergentes, y el intentar predecir y controlar estas propiedades emergentes es importante en el proceso de desarrollo del sistema. Debido a que los componentes son opacos, predecir sus propiedades emergentes es particularmente difícil. Como consecuencia, es posible encontrarse con que, cuando se integran componentes, el sistema resultante tiene propiedades no deseadas que limitan su uso.
4. *Equilibrio de requerimientos.* Normalmente se tiene que encontrar un equilibrio entre los requerimientos ideales y los componentes disponibles en el proceso de especificación y diseño del sistema. Por el momento, alcanzar este equilibrio es un proceso intuitivo. Necesitamos un método de análisis de equilibrio sistemático y más estructurado para ayudar a los diseñadores a seleccionar y configurar componentes.

Hasta ahora, el principal uso de CBSE ha sido la construcción de sistemas de informació de empresas, tales como sistemas de comercio electrónico. Los componentes que se reutil zan son desarrollados internamente o se obtienen de proveedores conocidos de confianz Aunque algunos vendedores venden componentes *online*, la mayoría de las compañías son t davía reticentes a confiar componentes binarios obtenidos de forma externa. No es probabl que la visión completa de CBSE con proveedores de componentes especializados tenga lug hasta que estos problemas que hemos mencionado puedan ser resueltos.

19.1 Componentes y modelos de componentes

Hay un consenso general en la comunidad de que un componente es una unidad de softwa independiente que puede estar compuesta por otros componentes y que se utiliza para cre un sistema software. Aparte de esto, sin embargo, distintas personas han propuesto defin ciones de un componente software. Councill y Heineman (Councill y Heineman, 2001) def nen un *componente* como:

> *Un elemento software que se ajusta a un modelo de componentes y que puede ser desplegado y compuesto de forma independiente sin modificación según un estándar de composición.*

Esta definición se basa fundamentalmente en estándares —una unidad software que ajusta a estos estándares es un componente—. Szyperski (Szyperski, 2002), sin embargo, menciona los estándares en su definición de componente, sino que, en vez de eso, se cent en las características clave de los componentes:

> *Un componente software es una unidad de composición con interfaces especificadas contractualmente y dependencias de contexto explícitas únicamente. Un componente software puede ser desplegado de forma independiente y está sujeto a la composición por terceras partes.*

Szyperski también constata que un componente no tiene un estado externamente observ ble. Esto significa que las copias de componentes son indistinguibles. Sin embargo, algun modelos de componentes, como el modelo Enterprise Java Beans, permite componentes c estado, lo que claramente no se corresponde con la definición de componente de Szypersk Mientras que los componentes sin estado son ciertamente más sencillos de usar, consider mos que CBSE debería acomodarse tanto a los componentes con estado como a los comp nentes sin estado.

Lo que tienen en común estas definiciones es que están de acuerdo en que los compone tes son independientes y que son unidades de composición fundamentales en un sistema. De de nuestro punto de vista, una definición completa de componente puede derivarse desde ar bas propuestas. La Figura 19.1 muestra lo que puede considerarse que son característic esenciales de un componente para ser usado en CBSE.

Estas definiciones formales de componentes son bastante abstractas y no proporcion realmente una idea clara de lo que hace un componente. Una de las formas más útiles de co siderar un componente es como un proveedor de servicios independiente. Cuando un siste necesita algún servicio, llama a un componente para proporcionar dicho servicio sin tener cuenta dónde se está ejecutando el componente o qué lenguaje se ha utilizado para desarr llar el componente. Por ejemplo, un componente en un sistema de biblioteca proporciona

Características del componente	Descripción
Estandarizado	La estandarización de componentes significa que un componente usado en un proceso CBSE tiene que ajustarse a algún modelo estandarizado de componentes. Este modelo puede definir interfaces de componentes, metadatos de componentes, documentación, composición y despliegue.
Independiente	Un componente debería ser independiente, debería ser posible componerlo y desplegarlo sin tener que utilizar otros componentes específicos. En las situaciones en las que el componente necesita servicios proporcionados externamente, éstos deberían hacerse explícitos en una especificación de interfaz del tipo «requiere».
Componible	Para que un componente sea componible, todas las interacciones externas deben tener lugar a través de interfaces definidas públicamente. Además debe proporcionar acceso externo a la información sobre sí mismo, como por ejemplo a sus métodos y atributos.
Desplegable	Para ser desplegable, un componente debe ser independiente y debe ser capaz de funcionar como una entidad autónoma o sobre una plataforma de componentes que implemente el modelo de componentes. Esto normalmente significa que el componente es binario y que no tiene que compilarse antes de ser desplegado.
Documentado	Los componentes tienen que estar completamente documentados para que los usuarios potenciales puedan decidir si los componentes satisfacen o no sus necesidades. La sintaxis e, idealmente, la semántica de todas las interfaces de componentes tienen que ser especificadas.

igura 19.1
aracterísticas de
omponentes.

servicio de búsqueda que permite a los usuarios buscar diferentes catálogos de bibliotecas; un componente que convierta de un formato gráfico a otro (por ejemplo, TIFF a JPEG) proporciona un servicio de conversión de datos.

La visión de un componente como un proveedor de servicios resalta dos características críticas de un componente reutilizable:

1. El componente es una entidad ejecutable independiente. El código fuente no está disponible, por lo que el componente no tiene que ser compilado antes de que sea usado con otros componentes del sistema.
2. Los servicios ofertados por un componente están disponibles a través de una interfaz, y todas las interacciones son a través de esa interfaz. La interfaz del componente se expresa en términos de operaciones parametrizadas y su estado interno nunca se muestra.

Los componentes se definen por sus interfaces y, en los casos más generales, puede considerarse que tienen dos interfaces relacionadas, tal y como se muestra en la Figura 19.2.

1. Una interfaz *proporciona*, que define los servicios proporcionados por el componente. La interfaz que hemos denominado proporciona, fundamentalmente, es el API del componente. Define los métodos que pueden ser llamados por un usuario del componente. Las interfaces proporciona se indican con un círculo al final de una línea desde el icono del componente.

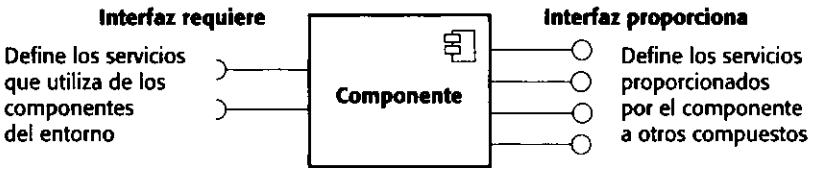


Figura 19.2 Interfaces de componentes.

2. Una interfaz *requiere*, que especifica qué servicios deben ser proporcionados por otro componentes en el sistema. Si éstos no están disponibles, entonces el componente n funcionará. Esto no compromete la independencia o el despliegue del componente de bido a que no se requiere el uso de un componente específico para proporcionar di chos servicios. Las interfaces requiere se indican con un semicírculo al final de una li nea desde el icono de componentes.

Por ejemplo, la Figura 19.3 muestra un modelo de un componente que ha sido diseñad para recopilar y comparar información de un vector de sensores. Se ejecuta de forma autóno ma para recoger datos durante un periodo de tiempo, y proporciona bajo demanda la compa ración de los datos a otro componente. La interfaz proporciona incluye métodos para añadi eliminar, iniciar, parar y probar los sensores. También incluye métodos para informes (repo y **listAll**) que informan sobre los datos recopilados y la configuración de los sensores. Aunqu no se ha mostrado esto aquí, estos métodos naturalmente tienen asociados parámetros que es pecifican la localización de los sensores y otra información.

El componente de recopilación de datos requiere que los sensores proporcionen una inter faz de gestión y una interfaz de datos. Éstos tienen parámetros que especifican el funciona miento y los datos que hay que recopilar. Se ha diseñado de forma deliberada la interfaz re querida para que no incluya operaciones específicas tales como **Test**. La interfaz requiere, má abstracta, permite al componente de recopilación de datos utilizarse con sensores con dife rentes interfaces. Un componente adaptador se utiliza como una interfaz entre el recopilado de datos y el interfaz hardware específico del sensor.

Las clases de objetos tienen métodos asociados que son claramente similares a los méto dos definidos en las interfaces de componentes. ¿Cuál es, entonces, la distinción entre com ponentes y objetos? Los componentes se desarrollan normalmente utilizando una aproxima ción orientada a objetos, pero difieren de los objetos en varios aspectos importantes:

1. *Los componentes son entidades desplegables*. Es decir, no son compilados en un prq grama de aplicación, sino que se instalan directamente sobre una plataforma de eje cución. Los métodos y atributos definidos en sus interfaces pueden ser accedidos pq otros componentes.

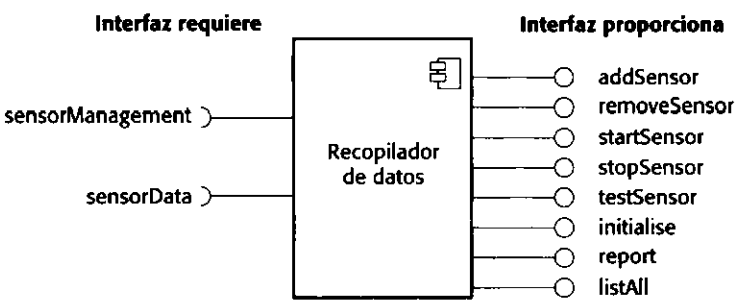


Figura 19.3 Un modelo de un componente de recopilación de datos.

2. *Los componentes no definen tipos.* Una definición de clase define un tipo abstracto de datos y los objetos son instancias de ese tipo. Un componente es una instancia, no una plantilla que se utiliza para definir una instancia.
3. *Las implementaciones de los componentes son opacas.* Los componentes están, al menos en principio, completamente definidos por la especificación de su interfaz. La implementación está oculta para los usuarios de los componentes. Los componentes a menudo se entregan como unidades binarias de forma que el comprador no tiene acceso a la implementación.
4. *Los componentes son independientes del lenguaje.* Las clases de objetos tienen que seguir las reglas de un lenguaje de programación orientado a objetos particular y, generalmente, sólo pueden interoperar con otras clases escritas en dicho lenguaje. Si bien los componentes se implementan normalmente utilizando lenguajes orientados a objetos tales como Java, pueden implementarse en lenguajes de programación no orientados a objetos.
5. *Los componentes están estandarizados.* A diferencia de las clases de objetos que pueden implementarse de cualquier forma, los componentes deben ajustarse a algún modelo de componentes que restringe su implementación.

19.1.1 Modelos de componentes

Un modelo de componentes es una definición de los estándares para la definición de componentes, documentación y despliegue. Estos estándares son utilizados por los desarrolladores de componentes para asegurar que los componentes puedan interoperar. También son utilizados por los proveedores de infraestructuras de comunicación de los componentes, quienes proporcionan middleware para soportar el funcionamiento de éstos. Se han propuesto muchos modelos de componentes, pero los modelos más importantes son el modelo de componentes CORBA de OMG, el modelo Enterprise Java Beans de Sun y el modelo COM+ de Microsoft (Blevins, 2001; Ewald, 2001; Wang *et al.*, 2001).

Las tecnologías de infraestructura específicas tales como COM+ y EJB que se utilizan en CBSE son muy complejas. Como consecuencia, es difícil describir estas tecnologías sin hacer referencia a una gran cantidad de detalles de implementación sobre los supuestos que subyacen en cada aproximación y las interfaces utilizadas. En lugar de entrar en estos detalles aquí, nos centramos en los elementos fundamentales de los modelos de componentes.

Los elementos básicos de un modelo de componentes ideal son analizados por Weinreich y Sametinger (Weinreich y Sametinger, 2001). Estos elementos del modelo se resumen en la Figura 19.4. Este diagrama muestra que los elementos en un modelo de componentes pueden clasificarse como elementos relacionados con las interfaces de los componentes, elementos relacionados con la información que se necesita para utilizar el componente en un programa y elementos relacionados con el despliegue del componente.

Los elementos que definen a un componente son sus interfaces. El modelo de componentes especifica cómo deberían definirse las interfaces y los elementos, tales como nombres de operaciones, parámetros y excepciones, que deberían incluirse en la definición de una interfaz. El modelo también debería especificar el lenguaje utilizado para definir las interfaces (el IDL). En CORBA y COM+, existe un lenguaje de definición de interfaces específico; EJB es específico de Java, por lo que Java se utiliza como IDL. Algunos modelos de componentes requieren interfaces específicas que deben ser definidas por un componente. Éstas se utilizan para integrar el componente con la infraestructura del modelo de componentes que proporciona servicios estandarizados tales como seguridad y gestión de transacciones.

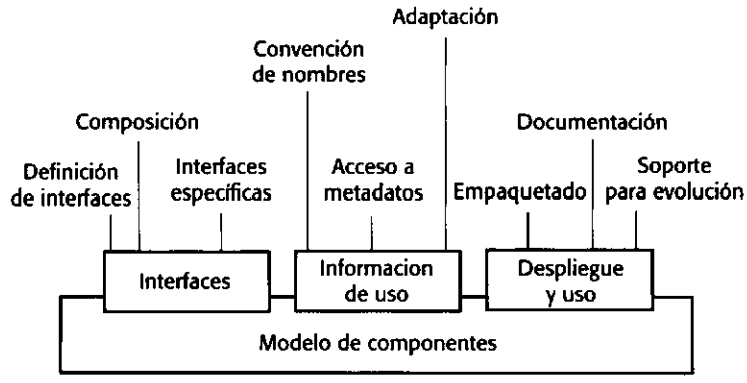


Figura 19.4 Elementos básicos de un modelo de componentes.

Para que los componentes puedan distribuirse y ser accedidos de forma remota, necesitan tener un nombre o manejador único asociado a ellos. En COM+, éste es un identificador único de 128 bits. En el modelo de componentes CORBA y en EJB, es un nombre jerárquico con una raíz basado en un nombre de dominio de Internet. Los metadatos de componentes son datos sobre el mismo componente, tales como información sobre sus interfaces y atributos. Los metadatos son importantes para que los usuarios del componente puedan encontrar qué servicios se proporcionan y cuáles se requieren. Las implementaciones del modelo de componentes normalmente incluyen formas específicas (como el uso de una interfaz de reflexión en Java) para acceder a estos metadatos del componente.

Los componentes son entidades genéricas y, cuando son desplegados, tienen que ser configurados para su entorno de aplicación particular. Por ejemplo, el componente **Recopilación de datos** mostrado en la Figura 19.3 podría ser configurado con el máximo número de sensores en un vector de sensores. Por lo tanto, el modelo de componentes debería especificar cómo pueden configurarse los componentes binarios para un entorno de despliegue particular.

Una parte importante de un modelo de componentes es una definición de cómo los componentes deberían empaquetarse para su despliegue como entidades ejecutables independientes. Debido a que los componentes son entidades independientes, tienen que ser empaquetadas con el resto de los elementos que no son proporcionados por la infraestructura del componente o no están definidos en una interfaz requiere. La información de despliegue incluye información sobre los contenidos de un paquete y su organización binaria.

Inevitablemente, a medida que surgen nuevos requerimientos, los componentes tendrán que ser cambiados o reemplazados. Por lo tanto, el modelo de componentes debería incluir reglas para regular cuándo y cómo se permite el reemplazo de componentes. Finalmente, el modelo de componentes debería definir la documentación asociada a los componentes. Ésta se utiliza para encontrar el componente y decidir si es adecuado o no.

Los modelos de componentes no son sólo estándares; son también la base para el middleware de sistemas que proporciona el soporte para los componentes ejecutables. Weinreich y Sametinger (Weinreich y Sametinger, 2001) utilizan la analogía de un sistema operativo para explicar los modelos de componentes. Un sistema operativo proporciona un conjunto de servicios genéricos que pueden utilizarse por las aplicaciones. Una implementación de un modelo de componentes proporciona servicios compartidos comparables para los componentes. La Figura 19.5 muestra algunos de los servicios que pueden ser proporcionados por una implementación de un modelo de componentes.

Los servicios proporcionados por la implementación de un modelo de componentes pueden pertenecer a dos categorías:

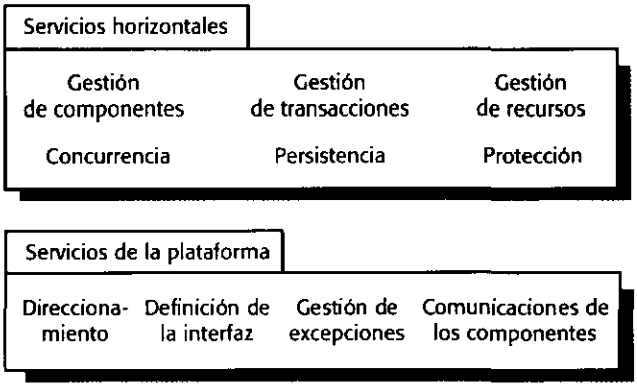


Figura 19.5 Servicios proporcionados por un modelo de componentes.

1. *Servicios de plataforma*. Estos servicios fundamentales permiten a los componentes comunicarse entre sí. CORBA es un ejemplo de una plataforma de modelos de componentes. Los servicios de plataforma se describen en el Capítulo 12.
2. *Servicios horizontales*. Estos servicios independientes de la aplicación es probable que sean usados por muchos componentes diferentes. La disponibilidad de estos servicios reduce los costes del desarrollo de componentes e implica que pueden evitarse incompatibilidades potenciales de componentes.

Para hacer uso de los servicios proporcionados por una infraestructura de modelos de componentes, los componentes se despliegan en un contenedor estandarizado predefinido. Un *contenedor* es un conjunto de interfaces utilizadas para acceder a las implementaciones de los servicios de soporte. La inclusión del componente en el contenedor proporciona automáticamente el acceso a los servicios. Las interfaces del componente en sí mismas no son accesibles directamente por otros componentes; tienen que ser accedidas a través del contenedor.

19.1.2 Desarrollo de componentes para reutilización

La visión a largo plazo de CBSE es que habrá proveedores de componentes cuyos negocios se basen en el desarrollo y venta de componentes reutilizables. Como ya se ha indicado, los problemas de confianza implican que hasta ahora no haya sido desarrollado un mercado abierto de componentes, y la mayoría de los componentes que son reutilizados han sido desarrollados dentro de la empresa. Los componentes reutilizables no están especialmente desarrollados, sino que se basan en componentes existentes que ya han sido implementados y utilizados en sistemas de aplicaciones.

En general, los componentes desarrollados internamente no son inmediatamente reutilizables. Incluyen características específicas de la aplicación e interfaces que son poco probables que sean requeridas en otras aplicaciones. Por consiguiente, hay que adaptar y extender estos componentes para crear una versión más genérica y por lo tanto más reutilizable. Obviamente esto tiene un coste asociado. Usted tiene que decidir, en primer lugar, si un componente va a ser probablemente reutilizado y en segundo lugar, si el ahorro de costes de la reutilización justifican los costes para hacer que ese componente sea reutilizable.

Para responder a la primera de estas cuestiones, tiene que decidirse si el componente implementa una o más abstracciones del dominio estables. Las abstracciones del dominio estables son conceptos fundamentales en el dominio de aplicaciones que cambian muy lentamente. Por ejemplo, en un sistema bancario, las abstracciones del dominio podrían incluir

cuentas, depósitos de cuentas y extractos. En un sistema de gestión de un hospital, las abstracciones del dominio podrían incluir pacientes, tratamientos y enfermeras. Estas abstracciones del dominio se denominan a veces objetos de empresa. Si el componente es una implementación de un objeto de empresa o grupo de objetos relacionados utilizados con frecuencia, probablemente pueden ser reutilizados.

Para responder a la cuestión sobre la efectividad del coste, tienen que evaluarse los costes de los cambios requeridos para hacer que el componente sea reutilizable. Estos costes son los costes de la documentación del componente, la validación del componente y los costes para hacer que el componente sea más genérico. Los cambios que pueden hacerse para que un componente sea más reutilizable incluyen:

- Eliminar los métodos específicos de la aplicación.
- Cambiar los nombres para hacerlos más generales.
- Añadir métodos para proporcionar una cobertura funcional más completa.
- Hacer que el manejo de excepciones sea consistente para todos los métodos.
- Añadir una interfaz de «configuración» para permitir que el componente se adapte a diferentes situaciones de uso.
- Integrar los componentes requeridos para incrementar la independencia.

El problema del manejo de excepciones es particularmente difícil. En principio, todas las excepciones deberían formar parte de la interfaz del componente. Los componentes no deberían manejar las excepciones por sí mismos. En su lugar, el componente debería definir qué excepciones pueden ocurrir y debería publicarlas como parte de su interfaz. Por ejemplo, un sencillo componente que implemente una estructura de pila de datos debería detectar y hacer públicas las excepciones de desbordamiento y la falta de elementos. En la práctica, sin embargo, un componente puede proporcionar algún manejo de excepciones local, y cambiar esto puede tener implicaciones serias para la funcionalidad del componente.

Mili y otros (Mili *et al.*, 2002) plantean formas de estimar los costes para hacer que un componente sea reutilizable y estimar el resultado de esta inversión. Los beneficios de reutilizar en lugar de redesarrollar un componente no son simples ganancias en cuanto a productividad. También pueden incluir ganancias en cuanto a calidad, debido a que el componente reutilizado suele ser más confiable, y ganancias en cuanto a oportunidad de mercado. También se pueden tener en cuenta las ganancias como consecuencia de desplegar el software más rápidamente. Mili y otros presentan varias fórmulas para estimar estas ganancias, al igual que lo hace el modelo COCOMO descrito en el Capítulo 26 (Boehm *et al.*, 2000). Sin embargo, los parámetros de estas fórmulas son difíciles de estimar con exactitud, y las fórmulas deben adaptarse a circunstancias locales. Se presume que estos factores conllevan que muy pocos gestores de proyectos software estén dispuestos a confiar en ellas.

Obviamente, si un componente es reutilizable o no, depende de su dominio de aplicación y funcionalidad. A medida que se añade generalidad a un componente, se incrementa su reusabilidad. Sin embargo, esto normalmente implica que el componente tenga más operaciones y sea más complejo, lo que hace que el componente sea más difícil de comprender y utilizar.

Existe un desequilibrio inevitable entre reusabilidad y la usabilidad de un componente. Hacer que el componente sea reutilizable implica proporcionar una serie de interfaces genéricas con operaciones que abarcan todas las formas en las cuales el componente podría ser utilizado. Hacer que el componente sea usable significa proporcionar una interfaz mínima sencilla que sea fácil de comprender. La reusabilidad añade complejidad y, por eso, reduce la comprensibilidad del componente. Por lo tanto, es más difícil decidir cuándo y cómo reutilizar ese

componente. Cuando se diseña un componente reutilizable, se debe buscar un equilibrio entre generalidad y comprensibilidad.

Otra importante fuente de componentes son los sistemas heredados existentes. Tal y como se expuso en el Capítulo 2, hay sistemas que desempeñan una función importante para la empresa, pero están escritos utilizando tecnologías software obsoletas. Debido a esto, puede ser difícil utilizarlos con nuevos sistemas. Sin embargo, si se convierten estos sistemas antiguos en componentes, su funcionalidad puede reutilizarse en aplicaciones nuevas.

Desde luego, estos sistemas heredados normalmente no tienen definidas claramente las interfaces requiere y proporciona. Para hacer que estos componentes sean reutilizables, tienen que definirse las interfaces del componente mediante lo que se conoce como *wrapper*. El wrapper oculta la complejidad del código subyacente y proporciona una interfaz para que los componentes externos puedan acceder a los servicios que dicho código proporciona. Naturalmente, este wrapper es un elemento de software bastante complejo ya que tiene que acceder a la funcionalidad del sistema heredado. Sin embargo, el coste del desarrollo de un wrapper es a menudo mucho menor que el coste de reimplementar el sistema heredado.

19.2 El proceso CBSE

Se ha sugerido en la introducción que la reutilización con éxito de componentes requiere un proceso de desarrollo adaptado a CBSE. La estructura de dicho proceso se analizó en el Capítulo 4; la Figura 19.6 muestra las principales subactividades dentro de un proceso CBSE. Algunas de las actividades dentro de este proceso, tales como el descubrimiento inicial de los requerimientos del usuario, se llevan a cabo de la misma forma que en otros procesos software. Sin embargo, las diferencias fundamentales entre este proceso y los procesos software basados en el desarrollo original del software son las siguientes:

1. Los requerimientos del usuario se desarrollan inicialmente en forma de esquema en lugar de con detalle, y los stakeholders son alentados a ser lo más flexibles posible en la definición de sus requerimientos. La razón de esto es que los requerimientos muy específicos limitan el número de componentes que podrían satisfacer estos requerimientos. Sin embargo, y a diferencia del desarrollo incremental, se necesita un conjunto completo de requerimientos con el fin de que se puedan identificar para su reutilización tantos componentes como sea posible.

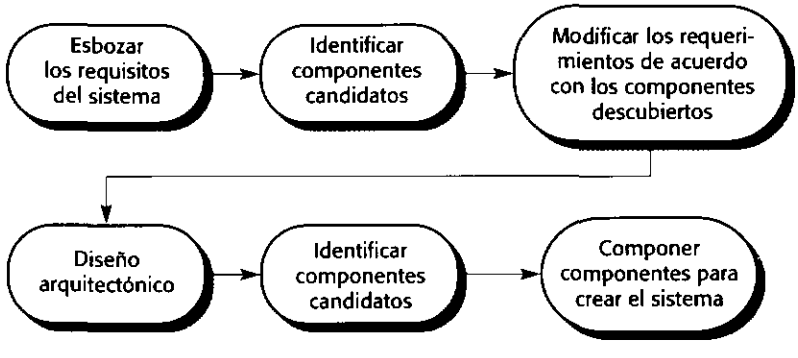


Figura 19.6
El proceso CBSE.

2. Los requerimientos son refinados y modificados en etapas tempranas en el proceso dependiendo de los componentes disponibles. Si los requerimientos del usuario no pueden ser satisfechos por los componentes disponibles, deberían discutirse los requerimientos relacionados que pueden ser soportados. Los usuarios pueden estar dispuestos a cambiar sus ideas si esto significa entregar el sistema más rápidamente y con un menor coste económico.
3. Hay una actividad adicional de búsqueda de componentes y refinamiento de diseño después de que la arquitectura del sistema haya sido diseñada. Algunos componentes aparentemente usables pueden resultar no adecuados o pueden no trabajar adecuadamente con otros componentes seleccionados. Aunque no se muestra en la Figura 19.6, esto implica que pueden ser necesarios cambios adicionales en los requerimientos.
4. El desarrollo es un proceso de composición en el que se integan los componentes descubiertos. Esto implica la integración de componentes con la infraestructura del modelo de componentes y, a menudo, desarrollar «código pegamento» para conciliar las interfaces de los componentes incompatibles. Por supuesto, puede requerirse funcionalidad adicional por encima y por debajo de la proporcionada por los componentes usables. Naturalmente, se debería desarrollar esto como componentes que pueden ser reutilizados en sistemas futuros.

La etapa de diseño arquitectónico es particularmente importante. Durante el diseño arquitectónico, se puede optar finalmente por un modelo de componentes, aunque, para muchos sistemas, esta decisión se hará antes de que comience la búsqueda de componentes. Tal y como se ha explicado en los Capítulos del 11 al 13, también se establece la organización de alto nivel del sistema y se toman decisiones sobre la distribución del sistema y el control. Jacobsen y otros (Jacobsen *et al.*, 1997) afirman que definir una arquitectura robusta es crítico para una reutilización exitosa.

Una actividad que es única para el proceso CBSE es la identificación de componentes. Ésta implica varias subactividades, tal y como se muestra en la Figura 19.7. Hay dos etapas en el proceso CBSE en las que usted tiene que identificar componentes para su posible uso en el sistema. En etapas tempranas del desarrollo, la atención debería centrarse en la búsqueda y selección. Es necesario convencerse a sí mismo de que hay componentes disponibles que satisfacen los requerimientos. Obviamente, debería realizarse alguna comprobación inicial de que el componente es adecuado, pero puede que no se necesiten pruebas detalladas. En etapas posteriores, después de que se haya diseñado la arquitectura del sistema, debería invertirse más tiempo en la validación de componentes. Hay que asegurarse de que los componentes identificados son realmente adecuados para su aplicación; si no, tienen que repetirse los procesos de búsqueda y selección.

La primera etapa en la identificación de componentes es la búsqueda de componentes que están disponibles localmente o son de proveedores de confianza. La visión de los defensores de CBSE como Szyperski (Szyperski, 2002) es que debería haber un mercado de componentes viable en donde los vendedores externos compitan para proporcionar componentes. En el momento de escribir este libro, esto no es una realidad. La principal razón para ello es que los usuarios de componentes externos se enfrentan con los riesgos de que estos componentes no

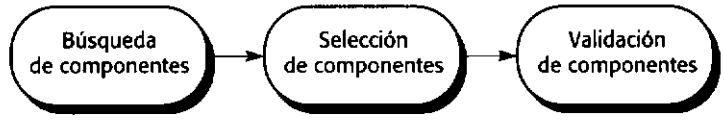


Figura 19.7 El proceso de identificación de componentes.

funcionen tal y como se les dice. Si éste es el caso, los costes de reutilización superan a los beneficios, y pocos gestores de proyectos creen que los riesgos merecen la pena. Otra importante razón de por qué los mercados de componentes no se han desarrollado es que muchos componentes están especializados en dominios de aplicaciones. No hay un mercado suficientemente grande en estos dominios para que los suministradores de componentes externos puedan establecer un negocio viable a largo plazo.

Como consecuencia, la búsqueda de componentes a menudo se ve restringida a la misma organización que desarrolla el software. Las compañías de desarrollo de software pueden formar su propia base de datos de componentes reutilizables sin los riesgos inherentes al uso de componentes de suministradores externos.

Una vez que el proceso de búsqueda de componentes ha identificado componentes candidatos, tienen que seleccionarse componentes específicos a partir de esta lista. En algunos casos, ésta será una tarea sencilla. Los componentes de la lista se corresponderán directamente con los requerimientos del usuario, y no habrá componentes que compitan para satisfacer estos requerimientos. Sin embargo, en otros casos, el proceso de selección es mucho más complejo. No habrá una clara correspondencia entre requerimientos y componentes, y se observará que tienen que utilizarse varios componentes para satisfacer un requerimiento específico o grupo de requerimientos. Desgraciadamente, es probable que diferentes requerimientos requieran diferentes grupos de componentes, por lo que habrá que decidir qué composición de componentes proporcionan el mejor cumplimiento de los requerimientos.

Una vez que se han seleccionado los componentes para su posible inclusión en un sistema, habría que validarlos para comprobar que se comportan como deberían. El alcance de validación requerido depende de la fuente de los componentes. Si se está usando un componente que ha sido desarrollado por una fuente conocida de confianza, puede decidirse qué pruebas de componentes independientes no son necesarias, y se prueba el componente cuando se integra con otros componentes. Por otro lado, si se está utilizando un componente proveniente de una fuente desconocida, siempre se debería comprobar y verificar ese componente antes de incluirlo en el sistema.

La validación de componentes implica desarrollar un conjunto de casos de prueba para el componente (o, posiblemente, extender los casos de prueba proporcionados con el componente) y desarrollar software adicional de pruebas para ejecutar las pruebas de componentes. El principal problema con la validación de componentes es que la especificación del componente puede no ser lo suficientemente detallada para permitir desarrollar un conjunto completo de pruebas de componentes. Los componentes se especifican normalmente de manera informal, siendo la única documentación formal la especificación de su interfaz. Ésta puede no incluir suficiente información para desarrollar un conjunto completo de pruebas que podrían llevar a la convicción de que la interfaz de dicho componente es la que se requiere.

Un problema adicional de validación, que podría ocurrir en esta etapa, es que el componente puede tener características que interfieran en el uso que se hace del componente. Los componentes reutilizables a menudo tendrán más funcionalidad de la que se necesita. Se puede simplemente prescindir de la funcionalidad que no se necesita, pero ésta puede a veces interferir en otros componentes o en el sistema en su totalidad. En algunos casos, la funcionalidad no deseada puede incluso causar fallos graves del sistema. La Figura 19.8 describe brevemente una situación en la que la funcionalidad no necesaria en un sistema reutilizado provocó un fallo catastrófico del software.

El problema en la lanzadera del Ariane 5 ocurrió debido a que las suposiciones hechas para el Ariane 4 no eran válidas para el Ariane 5. Éste es un problema general que presentan los componentes reutilizables. Éstos son implementados originalmente para un entorno de apli-

El fallo en la lanzadera del Ariane 5

Mientras se desarrollaba la lanzadera espacial del Ariane 5, los diseñadores decidieron reutilizar el software de referencia inercial que había funcionado con éxito en la lanzadera del Ariane 4. El software de referencia inercial mantiene la estabilidad del cohete. Decidieron reutilizarlo sin cambios (tal y como se haría con los componentes), aunque se incluyeron funcionalidades adicionales que eran necesitaban en el Ariane 5.

En el primer lanzamiento del Ariane 5, el software de navegación inercial falló después de 37 segundos y el cohete no pudo ser controlado. Los controladores de tierra dieron órdenes a la lanzadera para autodestruirse y la carga del Ariane fue destruida. Una investigación subsiguiente descubrió que la causa del problema fue una excepción no manejada cuando una conversión desde un número de punto fijo a un entero dio como resultado un desbordamiento numérico. Esto hizo que el sistema de ejecución abortara el sistema de referencia inercial y la estabilidad de la lanzadera no pudo mantenerse. El fallo nunca ocurrió en el Ariane 4 debido a que tuvo motores con menos potencia y el valor que fue convertido no era tan grande como para provocar un desbordamiento en la conversión.

El fallo se produjo en el código que no era requerido en el Ariane 5. Las pruebas de validación para el software reutilizado se basaron en los requerimientos del Ariane 5. Debido a que no había requerimientos para la función que falló, no se desarrollaron pruebas. Consecuentemente, el problema con el software nunca fue descubierto durante las pruebas de simulación de la lanzadera.

Figura 19.8 Un fallo en la validación de un componente.

cación y, naturalmente, incluyen suposiciones sobre dicho entorno. Estas suposiciones son raramente documentadas y, por lo tanto, cuando el componente es reutilizado, es imposible derivar pruebas para comprobar si las suposiciones son todavía válidas.

19.3 Composición de componentes

La composición de componentes es el proceso de ensamblar componentes para crear un sistema. Si nosotros asumimos una situación en la que los componentes reutilizables están disponibles, entonces la mayoría de los sistemas se construirán componiendo estos componentes reutilizables unos con otros, con componentes escritos de forma especial y con la infraestructura de soporte de los componentes proporcionada por el marco de trabajo del modelo. Tal y como se ha indicado en la Sección 19.1, esta infraestructura proporciona facilidades para soportar la comunicación de componentes y los servicios horizontales como los servicios de interfaz de usuario, gestión de transacciones, concurrencia y protección. Las formas en las que los componentes se integran con esta infraestructura son documentadas para cada modelo de componentes y no se tratan en esta sección.

La composición no es una operación sencilla; existen varios tipos de composiciones (Figura 19.9):

1. *Composición secuencial.* Tiene lugar cuando, en el componente compuesto, los componentes constituyentes se ejecutan en secuencia. Se corresponde con la situación (a) en la Figura 19.9, en donde se componen las interfaces proporciona de cada componente. Se requiere algún código extra para conseguir enlazar los componentes.

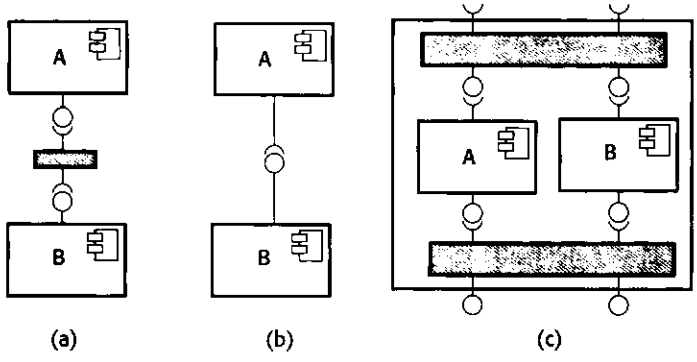


Figura 19.9 Tipos de composición de componentes.

2. *Composición jerárquica*. Tiene lugar cuando un componente realiza una llamada directamente a los servicios proporcionados por otro componente. Se corresponde con una situación en que la interfaz proporciona de un componente se compone con la interfaz requiere de otro componente. Ésta es la situación (b) en la Figura 19.9.
3. *Composición aditiva*. Tiene lugar cuando las interfaces de dos o más componentes se unen (se añaden) para crear un nuevo componente. Las interfaces del componente compuesto se crean uniendo todas las interfaces de los componentes constituyentes, eliminando las operaciones duplicadas si es necesario. Esto se corresponde con la situación (c) de la Figura 19.9.

Podrían utilizarse todas las formas de composición de componentes al crear un sistema. En todos los casos, se puede tener que escribir «código pegamento» que enlaza los componentes. Por ejemplo, para la composición secuencial, la salida de un componente A normalmente se convierte en la entrada para el componente B. Se necesitan sentencias intermedias que llamen al componente A, recojan el resultado y a continuación llamen al componente B con dicho resultado como un parámetro.

Cuando se escriben componentes especialmente para composición, se diseñan las interfaces de dichos componentes para que sean compatibles. Por lo tanto, se pueden componer fácilmente estos componentes en una única unidad. Sin embargo, cuando los componentes se desarrollan de forma independiente para reutilización, se observarán a menudo incompatibilidades de interfaces en las que las interfaces de los componentes que se desea componer no son las mismas. Pueden darse tres tipos de incompatibilidades:

1. *Incompatibilidad de parámetros*. Las operaciones a cada lado de la interfaz tienen el mismo nombre, pero los tipos de los parámetros o su número son diferentes.
2. *Incompatibilidad de operaciones*. Los nombres de las operaciones en las interfaces proporciona y requiere son distintos.
3. *Operaciones incompletas*. La interfaz proporciona de un componente es un subconjunto de la interfaz requiere de otro componente o viceversa.

En todos los casos, el problema de la incompatibilidad se trata escribiendo un componente adaptador que reconcilia las interfaces de los dos componentes que se están reutilizando. Cuando se conocen las interfaces de los componentes que se desea utilizar, se describe un componente adaptador que convierte una interfaz en otra. La forma precisa del adaptador depende del tipo de composición. Algunas veces, como en el siguiente ejemplo, el adaptador simplemente toma un resultado de un componente y lo convierte en una forma tal que pueda ser usada como entrada por otro. En otros casos, el adaptador puede ser

llamado por el componente A y él mismo llama al componente B. La última situación podría ocurrir si A y B fuesen compatibles pero el número de parámetros en sus interfaces fuera diferente.

Para ilustrar los adaptadores, consideremos los componentes mostrados en la Figura 19.10. Éstos podrían ser parte de un sistema utilizado por los servicios de emergencia. Cuando el operador de la emergencia recibe una llamada, el número de teléfono es la entrada para el componente **addressFinder** para localizar la dirección. A continuación, utilizando el componente **mapper**, se imprime un mapa para ser enviado al vehículo asignado a la emergencia. De hecho, el componente podría tener interfaces más complejas que las mostradas aquí, pero la versión simplificada ilustra el concepto de un adaptador.

El primer componente, **addressFinder**, encuentra la dirección que se corresponde con un número de teléfono. También puede devolver el dueño de la propiedad asociada con el número de teléfono y el tipo de propiedad. El componente **mapper** toma un código postal (en los Estados Unidos, un código estándar ZIP con los cuatro dígitos adicionales que identifican la localización de la propiedad) y visualiza o imprime un mapa de calles del área alrededor de dicho código a una escala especificada.

Estos componentes son componibles en principio debido a que la propiedad de localización incluye el código postal o ZIP. Sin embargo, tiene que escribirse un componente adaptador llamado **postCodeStripper** que toma los datos de la localización de **addressFinder** y extrae el código postal. A continuación, este código postal se utiliza como entrada para **mapper**, y el mapa de calles se visualiza a una escala de 1:10.000. El siguiente código ilustra la secuencia de llamadas requeridas para implementar esto:

```
address = addressFinder.location(phonenumber);
postCode = postCodeStripper.getPostCode(address);
mapper.displayMap(postCode, 10000);
```

Otro caso en el que puede utilizarse un componente adaptador es cuando un componente desea hacer uso de otro, pero hay una incompatibilidad entre las interfaces proporciona y requiere de estos componentes. Esto se ilustra en la Figura 19.11, en donde el componente recolector de datos se conecta a un componente sensor utilizando un adaptador. Éste concilia las interfaces del componente recolector de datos con las interfaces proporciona del componente sensor. El componente recolector de datos fue diseñado con un mecanismo requiere genérico que no se basó en una interfaz de sensor específica. Ya se adelantó que un adaptador siempre podría ser utilizado para conectar el recolector de datos con una interfaz específica del sensor.

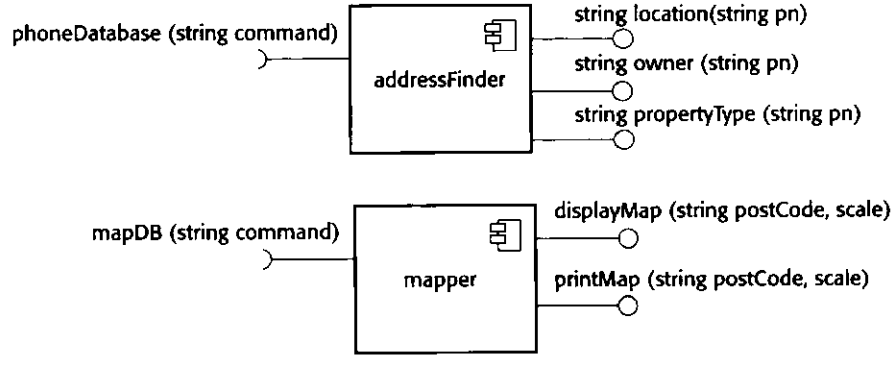


Figura 19.10
Componentes incompatibles.

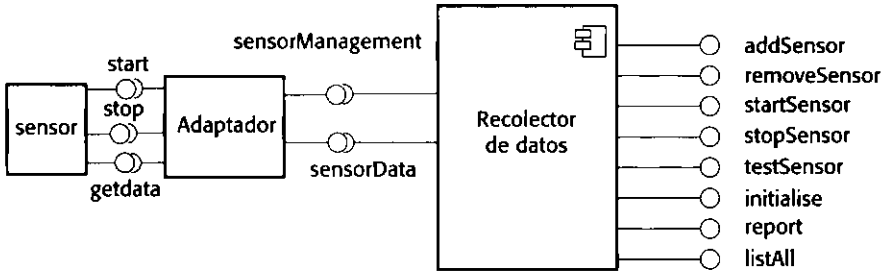


Figura 19.11
Adaptador para el componente recolector de datos.

La discusión sobre la composición de componentes supone que puede saberse a partir de la documentación de los componentes si las interfaces son compatibles. Por supuesto, la definición de la interfaz incluye el nombre de las operaciones y los tipos de los parámetros, de forma que puede evaluarse la compatibilidad a partir de éstas. Sin embargo, para decidir si las interfaces son semánticamente compatibles hay que partir de la documentación de los componentes.

Por ejemplo, consideremos la composición mostrada en la Figura 19.12. Estos componentes se utilizan para implementar un sistema que descarga imágenes de una cámara digital y las almacena en una librería fotográfica. El usuario del sistema puede proporcionar información adicional para describir y catalogar las fotografías. Para evitar confusiones, no se han mostrado aquí todos los métodos de las interfaces, sino que simplemente se muestran los métodos que son necesarios para ilustrar el problema de la documentación de los componentes. Los métodos en la interfaz del componente que se ha denominado **Photo Library** son:

```
public void addItem (Identifier pid; Photograph p; CatalogEntry photodesc);
public Photograph retrieve (Identifier pid);
public CatalogEntry catEntry (Identifier pid);
```

Se asume que la documentación para el método **addItem** del componente **Photo Library** es:

Este método añade una fotografía a la librería y asocia el identificador de la fotografía y el identificador del catálogo con la fotografía.

Esta descripción parece que es comprensible, pero considere las siguientes cuestiones:

¿Qué ocurre si el identificador de la fotografía ya está asociado con una fotografía en la librería?

¿Está también asociado el descriptor de la fotografía con la entrada del catálogo además de la fotografía? Es decir, si yo borro la fotografía, ¿también borro la información del catálogo?

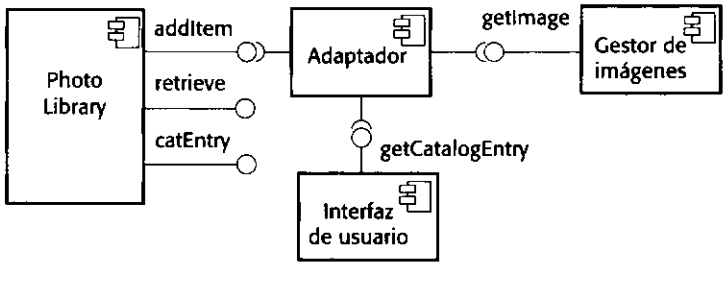


Figura 19.12
Composición de una librería fotográfica (Photo Library).

No hay suficiente información en la descripción informal de **addItem** para responder a estas cuestiones. Por supuesto, es posible añadir más información a la descripción en lenguaje natural del método, pero, en general, la mejor forma de resolver ambigüedades es utilizar un lenguaje formal para describir la interfaz. En el Capítulo 10, se sugirió que la descripción de la interfaz es un área en donde las especificaciones formales son bastante útiles. La especificación mostrada en la Figura 19.13 es parte de la descripción de la interfaz del componente **Photo Library** que añade información a la descripción informal.

La especificación en la Figura 19.13 utiliza pre y postcondiciones, y se ha empleado una notación basada en el lenguaje de restricciones de objetos (OCL) que forma parte de UML (Warmer y Kleppe, 1998). OCL se diseñó para describir restricciones en los modelos de objetos UML; permite expresar predicados que deben ser ciertos siempre, que deben ser ciertos antes de que un método sea ejecutado, y que deben ser ciertos después de que un método sea ejecutado. Éstos son los invariantes, precondiciones y postcondiciones respectivamente. Para acceder al valor de una variable antes de una operación, se añade **@pre** después de su nombre. Por lo tanto:

```
age = age@pre +1
```

significa que el valor de la variable age después de una operación es uno más del valor que tenía antes de dicha operación.

Las aproximaciones basadas en OCL se están utilizando cada vez más para añadir información semántica a los modelos UML. La aproximación general se ha derivado de la aproximación al Diseño mediante Contratos de Meyer (Meyer, 1992), en la cual las interfaces y obligaciones de comunicación de los objetos son especificadas formalmente y forzadas por el sistema de ejecución. Meyer sugiere que el uso de diseños por contratos es esencial si estamos desarrollando componentes fiables (Meyer, 2003).

```
-- La palabra clave context hace referencia al componente al que se aplican las
-- condiciones
context addItem

-- Las precondiciones especifican lo que debe ser cierto antes de la ejecución
-- de addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

-- Las postcondiciones especifican lo que es cierto después de la ejecución
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid)  null ;

post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre[em]1
```

Figura 19.13 Descripción formal de la interfaz de Photo Library.

La Figura 19.13 incluye una especificación para los métodos **addItem** y **delete** en el componente **Photo Library**. El método que se está especificando se indica con la palabra clave **context**, y las pre y postcondiciones con las palabras clave **pre** y **post**. Las precondiciones para **addItem** señalan que:

- No debería haber una fotografía en la librería con el mismo identificador que la nueva fotografía a añadir.
- La librería debe existir —se supone que la creación de una librería añade un único elemento a ésta de forma que el tamaño de una librería siempre es mayor que cero.

Las postcondiciones para **addItem** señalan que:

- El tamaño de la librería ha crecido en uno (de forma que solamente se ha efectuado una entrada).
- Si usted recupera utilizando el mismo identificador, entonces vuelve a la fotografía que usted añadió.
- Si usted busca en el catálogo utilizando ese identificador, vuelve a la entrada del catálogo que hizo.

La especificación de **delete** proporciona información adicional. La precondición indica que, para borrar un elemento, éste debe estar en la librería y, después de su borrado, la fotografía ya no puede ser recuperada y el tamaño de la librería se reduce en uno. Sin embargo, **delete** no borra la entrada del catálogo —todavía es posible recuperarla después de que la foto haya sido borrada—. La razón de esto es que puede quererse mantener información en el catálogo de por qué una foto fue borrada, su nueva localización, y así sucesivamente.

Cuando se crea un sistema mediante la composición de componentes, se puede observar que hay conflictos potenciales entre los requerimientos funcionales y no funcionales, la necesidad de entregar un sistema tan rápidamente como sea posible, y la necesidad de crear un sistema que pueda evolucionar a medida que los requerimientos cambian. Las decisiones en las que se tiene que llegar a un equilibrio son:

1. ¿Qué composición de componentes es más efectiva para satisfacer los requerimientos funcionales del sistema?
2. ¿Qué composición de componentes permite adaptaciones para futuros cambios sobre los requerimientos?
3. ¿Cuáles serán las propiedades emergentes del sistema compuesto? Estas propiedades emergentes son propiedades tales como rendimiento y confiabilidad. Solamente pueden evaluarse una vez que el sistema completo está implementado.

Desgraciadamente, hay muchas situaciones en las que las soluciones a los problemas de la composición están en conflicto mutuo. Por ejemplo, consideremos una situación tal como la que se ilustra en la Figura 19.14, en donde un sistema puede ser creado a través de dos alternativas de composiciones. El sistema es un sistema de informes y recolección de datos en el que los datos se recogen desde diferentes fuentes, se almacenan en una base de datos, y se elabora un informe que resume estos datos.

Las ventajas de la composición (a) son que la gestión e informes de datos son independientes, con lo que hay más flexibilidad para futuros cambios. El sistema de gestión de datos podría ser reemplazado y, si se requieren informes que el componente de informes actual no puede producir, ese componente también puede ser reemplazado. En la composición (b), se usa un componente de base de datos con facilidades de informes embebidos en ella (por ejem-

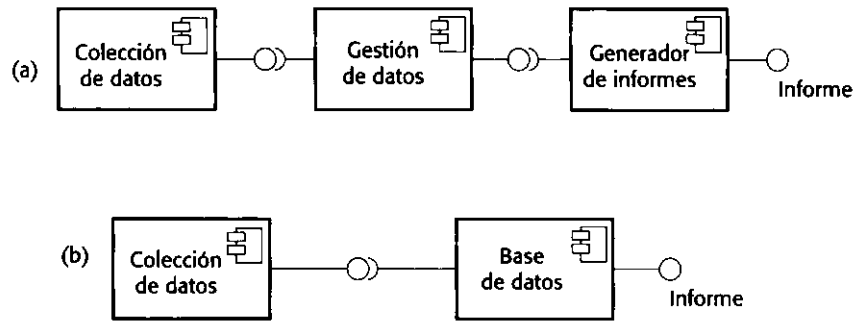


Figura 19.14 Componentes de recolección de datos y generación de informes.

plo Microsoft Access). Las ventajas de la composición (b) son que hay menos componentes, con lo que el sistema probablemente será más rápido debido a que no hay sobrecargas de comunicación entre los componentes. Además, las reglas de integridad de datos que se aplican a la base de datos también se aplicarán a los informes. Estos informes no podrán combinar datos de forma incorrecta. En la composición (a), no hay tales restricciones, por lo que los errores en los informes son más probables.

En general, un buen principio de composición a seguir es el principio de la separación de papeles. Es decir, debería intentarse diseñar el sistema de forma que cada componente tenga un papel claramente definido e, idealmente, estos papeles no deberían solaparse. Sin embargo, puede ser más económico construir un componente multifuncional en lugar de dos o tres componentes independientes. Además, el sistema puede sufrir penalizaciones en cuanto a confiabilidad o rendimiento cuando se utilizan múltiples componentes.

PUNTOS CLAVE

- **La ingeniería del software basada en componentes es una aproximación basada en la reutilización para definir, implementar y componer componentes independientes débilmente acoplados para formar sistemas.**
- **Un componente es una unidad software cuya funcionalidad y dependencias están completamente definidas por un conjunto de interfaces públicas. Los componentes pueden combinarse con otros componentes sin hacer referencia a su implementación y pueden ser desplegados como una unidad ejecutable.**
- **Un modelo de componentes define un conjunto de estándares para componentes, incluyendo estándares de interfaz, estándares de uso y estándares de despliegue. La implementación del modelo de componentes proporciona un conjunto de servicios horizontales que pueden ser utilizados por todos los componentes.**
- **Durante el proceso CBSE, tienen que intercalarse los procesos de ingeniería de requerimientos y diseño del sistema. Se tiene que alcanzar un equilibrio entre requerimientos deseables frente a servicios que están disponibles por componentes reutilizables existentes.**
- **La composición de componentes es el proceso de enlazar componentes para crear un sistema. Los tipos de composición incluyen composición secuencial, composición jerárquica y composición aditiva.**
- **Cuando se componen componentes reutilizables que no han sido creados explícitamente para la aplicación en la que se desea reutilizarlos, normalmente se necesitará escribir adaptadores o «código pegamento» para conciliar las diferentes interfaces de los componentes.**

■ **En la elección de composiciones, tiene que considerarse la funcionalidad requerida del sistema, los requerimientos no funcionales y la facilidad con la que un componente puede ser reemplazado por otro cuando el sistema cambia.**

LECTURAS ADICIONALES

Component-based Software Engineering: Putting the Pieces Together. Este libro es una colección de artículos de varios autores sobre diferentes aspectos de CBSE. Como todas las colecciones, es muy variado, pero estudia bastante mejor los aspectos generales de la ingeniería del software con componentes que el libro de Szyperski. (G. T. Heineman y W. T. Councill, 2001, Addison-Wesley.)

Component Software: Beyond Object-Oriented Programming, 2nd ed. Esta edición actualizada del primer libro sobre CBSE trata cuestiones técnicas y no técnicas en CBSE. Incluye más detalles sobre tecnologías específicas que el libro de Heineman y Councill y presenta un estudio sobre cuestiones de mercado. (C. Szyperski, 2002, Addison-Wesley.)

«Specification, Implementation and deployment of components». Una buena introducción a los fundamentos de CBSE. El mismo número de CACM incluye artículos sobre componentes y desarrollo basado en componentes [I. Crnkovic, *et al.*, *Comm. ACM*, 45(10), octubre, 2002.]

EJERCICIOS

19.1 ¿Por qué es importante que todas las interacciones entre componentes se definan a través de interfaces requiere y proporciona?

19.2 El principio de independencia de componentes implica que sea posible reemplazar un componente por otro que esté implementado de una forma completamente diferente. Usando un ejemplo, comente cómo tal reemplazo de componentes podría tener consecuencias no deseadas y podría conducir a un fallo de funcionamiento del sistema.

19.3 ¿Cuales son las diferencias fundamentales entre componentes y servicios web? (véase el Capítulo 12.)

19.4 ¿Por qué es importante que los componentes se basen en un modelo de componentes estándar?

19.5 Utilizando un ejemplo de un componente que implemente un tipo abstracto de datos tal como una pila o una lista, demuestre por qué normalmente es necesario extender y adaptar los componentes para su reutilización.

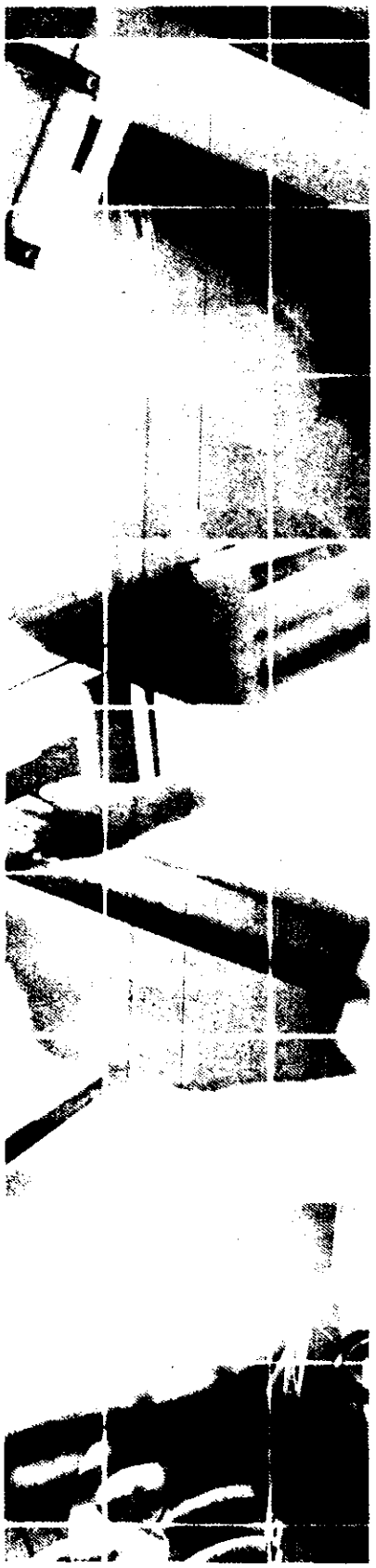
19.6 Explique por qué es muy difícil validar un componente reutilizable sin el código fuente del componente. ¿De qué formas podría una especificación formal de componentes simplificar el problema de validación?

19.7 Diseñe un componente reutilizable que implemente la característica de búsqueda del sistema LIBSYS descrito en capítulos anteriores. Éste no implementa una simple búsqueda por palabra clave de páginas web. Tiene que ser capaz de buscar los catálogos de varias bibliotecas, tal y como especifica el usuario.

19.8 Utilizando ejemplos, ilustre los diferentes tipos de adaptadores necesarios para soportar composición secuencial, composición jerárquica y composición aditiva.

19.9 Diseñe las interfaces de los componentes que podrían utilizarse en un sistema de una sala de control de emergencias. Usted debería diseñar interfaces para un componente de registro de llamadas que registre las llamadas realizadas, y un componente de búsqueda de vehículos que, dado un código postal y un tipo de incidente, encuentre el vehículo adecuado más cercano para ser enviado al lugar del incidente.

19.10 Se ha sugerido que podría establecerse una autoridad de certificación independiente. Los vendedores podrían someter sus componentes a esta autoridad, quien podría validar que el componente fuera fiable. ¿Cuáles podrían ser las ventajas y desventajas de dicha autoridad de certificación?



20 Desarrollo de sistemas críticos

Objetivos

El objetivo de este capítulo es introducir las técnicas de implementación utilizadas en el desarrollo de sistemas críticos. Cuando haya leído este capítulo:

- comprenderá cómo el evitar los defectos y la tolerancia a defectos contribuye al desarrollo de sistemas confiables;
- conocerá las características y las actividades de los procesos software confiables;
- habrá sido introducido en las técnicas de programación para evitar defectos;
- comprenderá las etapas implicadas en la implementación de la tolerancia a defectos y las formas en las que la diversidad y la redundancia son utilizadas en arquitecturas tolerantes a defectos.

Contenidos

Técnicas de ingeniería del software mejoradas, mejores lenguajes de programación y una mejor gestión de la calidad han contribuido a mejoras significativas en la confiabilidad de la mayoría del software. Sin embargo, los sistemas críticos, tales como aquellos que controlan maquinaria automática, sistemas médicos, centrales de telecomunicaciones o aviones necesitan niveles más altos de confiabilidad. En estos casos, pueden utilizarse técnicas de desarrollo especiales para asegurar que el sistema es seguro, protegido y fiable.

Existen dos aproximaciones complementarias para desarrollar software confiable:

1. *Prevención de defectos*. El proceso de diseño e implementación del sistema debería utilizar aproximaciones al desarrollo del software que ayuden a evitar errores de programación y así minimizar el número de defectos en un programa.
2. *Detección de defectos*. Los procesos de verificación y validación se diseñan para descubrir y validar defectos en un programa antes de que éste sea desplegado para su uso.
3. *Tolerancia a defectos*. El sistema se diseña para que los defectos o comportamientos inesperados del sistema durante la ejecución sean detectados y gestionados de tal forma que no ocurran fallos de funcionamiento.

Este capítulo se centra en las técnicas y procesos que soportan la prevención de defectos y la tolerancia a defectos. La detección de defectos es un tema importante por derecho propio y se trata en la Parte 5 de este libro. Se estudian técnicas estáticas para la detección de defectos en el Capítulo 22, las pruebas de programas en el Capítulo 23 y las técnicas de verificación y validación específicas para sistemas críticos en el Capítulo 24.

Resulta fundamental, para conseguir la confiabilidad en cualquier sistema, nociones básicas sobre redundancia y diversidad. La redundancia y la diversidad son estrategias cotidianas para evitar los fallos de ejecución. Si una persona invierte en bolsa de valores, no coloca todas sus inversiones en una única compañía, ya que podría perder todo si la compañía quiebra (diversidad). Las personas guardan baterías de reserva y bombillas en sus casas para poder recuperarse rápidamente de fallos eléctricos (redundancia). Nosotros deberíamos hacer copias de seguridad de nuestras computadoras de forma regular en caso de un fallo en el disco (redundancia) y, para asegurar nuestros hogares, a menudo tenemos más de un tipo de cerradura en la puerta (diversidad).

Los sistemas críticos pueden incluir componentes que reproducen la funcionalidad de otros componentes (redundancia) o código de comprobación adicional que no es estrictamente necesario para que el sistema funcione (redundancia). Por lo tanto, los defectos pueden detectarse antes de que ocasionen fallos de funcionamiento, y el sistema puede ser capaz de continuar funcionando si los componentes individuales fallan. Si los componentes redundantes del sistema no son exactamente iguales que otros componentes (diversidad), un fallo de funcionamiento en el componente duplicado tendrá como resultado un fallo de funcionamiento en el sistema completo.

En los sistemas en los que la disponibilidad es un requerimiento crítico, normalmente están disponibles servidores redundantes. Éstos automáticamente se ponen en funcionamiento si un determinado servidor falla. Algunas veces, para asegurar que los ataques sobre el sistema no puedan explotar alguna vulnerabilidad común, estos servidores pueden ser de diferentes tipos y ejecutarse sobre diferentes sistemas operativos. El uso de diferentes sistemas operativos es un ejemplo de diversidad y redundancia del software. En otros casos, tal y como se explica más adelante, la diversidad y la redundancia pueden ser embebidas en el software incluyendo componentes software redundantes que han sido programados de forma deliberada utilizando distintas técnicas.

La diversidad y la redundancia también se utilizan para conseguir procesos confiables. Al igual que cuando se prueba un programa, pueden utilizarse inspecciones de programas y análisis estático como técnicas para encontrar defectos. Estas técnicas de validación son complementarias: una puede encontrar defectos que son ocultados por otra. Además, la misma actividad del proceso (por ejemplo, una inspección de programa) puede ser llevada a cabo por varios miembros del grupo; las personas asumen tareas de diferentes formas dependiendo de su personalidad, experiencia y educación, de modo que este tipo de redundancia proporciona una perspectiva variada del sistema.

Por desgracia, añadir diversidad y redundancia a los sistemas los hace más complejos y, por lo tanto, más difíciles de entender. Por esta razón, es más probable que los programadores comentan errores y menos probable que las personas que prueben el programa encuentren errores. Como consecuencia, algunas personas piensan que es mejor evitar la redundancia y la diversidad en el software, y así diseñar el sistema para que sea lo más sencillo posible y permitir realizar procedimientos de verificación y validación con una rigurosidad extrema (Parnas *et al.*, 1990). Ambas aproximaciones se utilizan en sistemas de seguridad críticos comerciales. El sistema de control de vuelo del Airbus 340 es diverso y redundante (Storey, 1996), mientras que el sistema de control de vuelo del Boeing 777 se basa en una versión sencilla del software.

Un objetivo en la investigación en ingeniería del software ha sido el desarrollar herramientas, técnicas y métodos que lleven a la producción de software libre de defectos. El software libre de defectos es el software que cumple exactamente con su especificación. Sin embargo, esto no significa que el software nunca falle. Puede haber errores en la especificación que se reflejan en el software, o los usuarios pueden no comprender o no usar bien el sistema software. Sin embargo, eliminar los defectos del software ciertamente tiene un enorme impacto sobre el número de fallos de ejecución del sistema.

Para sistemas de tamaño pequeño y medio, nuestras técnicas de ingeniería del software son tales que probablemente sea posible desarrollar software libre de defectos. Para alcanzar este objetivo, es necesario utilizar varias técnicas de ingeniería del software:

1. *Procesos software confiables.* El uso de un proceso software confiable (analizado en la Sección 20.1) con actividades de verificación y validación adecuadas resulta esencial si tiene que minimizarse el número de defectos en el programa, y tienen que detectarse aquellos que se produzcan.
2. *Gestión de calidad.* La organización que desarrolla el sistema debe tener una cultura en la que la calidad guíe el proceso software. Esta cultura debería motivar a los programadores a escribir programas libres de errores. Deberían establecerse estándares de diseño y desarrollo, así como procedimientos para comprobar que dichos estándares se cumplen.
3. *Especificación formal.* Debe realizarse una especificación del sistema precisa (preferiblemente formal) que defina el sistema que se va a implementar. Muchos errores de diseño y programación son el resultado de una mala interpretación de una especificación ambigua o pobremente redactada.
4. *Verificación estática.* Las técnicas de verificación estáticas, como el uso de analizadores estáticos, pueden encontrar características anómalas en el programa que podrían ser defectos. También podría usarse verificación formal, basada en la especificación del sistema.
5. *Tipado fuerte.* Para el desarrollo se debe usar un lenguaje de programación fuertemente tipado, como Java o Ada. Si el lenguaje es fuertemente tipado, el compilador

del lenguaje puede detectar muchos errores de programación antes de que puedan ser introducidos en el programa entregado.

6. *Programación segura.* Algunas construcciones de lenguajes de programación son más complejas y propensas a error que otras, y es más probable que se cometan errores si se usan. La programación segura significa evitar, o al menos minimizar, el uso de estas construcciones.
7. *Información protegida.* Debería utilizarse una aproximación para el diseño e implementación del software basada en la ocultación y encapsulamiento de la información. Los lenguajes orientados a objetos como Java, obviamente satisfacen esta condición. Se debería fomentar el desarrollo de programas diseñados para ser legibles y comprensibles.

Se han comentado algunas de estas técnicas en otros capítulos del libro. Este capítulo se centra en la descripción de procesos software confiables y técnicas de programación que contribuyen a la prevención de defectos.

Sin embargo, hay situaciones difíciles en las que es económicamente práctico utilizar todas estas técnicas para crear software libre de defectos. El coste de encontrar y eliminar defectos en el sistema crece exponencialmente a medida que se descubren y eliminan los defectos en el programa (Figura 20.1). A medida que el software se hace más fiable, se necesita emplear más y más tiempo y esfuerzo para encontrar menos y menos defectos. En algún momento, los costes de este esfuerzo adicional se convierten en injustificables.

Como consecuencia, las compañías de desarrollo de software aceptan que su software siempre tendrá algún defecto residual. El nivel de defectos depende del tipo de sistema. Los productos relacionados con sistemas no críticos tienen un relativamente alto nivel de defectos (aunque son mucho mejores que hace diez años), mientras que los sistemas críticos normalmente tienen mucha menor densidad de defectos.

El razonamiento para aceptar defectos es que, si y cuando el sistema falla, es más barato pagar por las consecuencias de un fallo de funcionamiento que descubrir y eliminar los defectos antes de la entrega del sistema. Sin embargo, tal y como se explica en el Capítulo 3, la decisión de entregar software con defectos no es simplemente económica. La aceptabilidad social y política de un fallo de funcionamiento del sistema también ha de tenerse en cuenta.

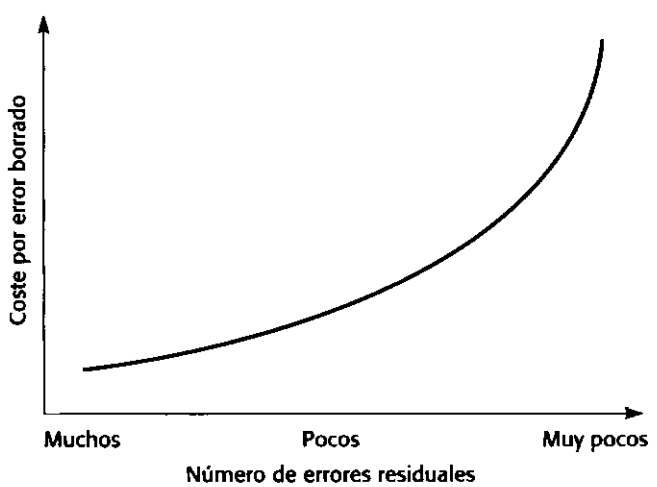


Figura 20.1 Costes crecientes de la eliminación de defectos residuales.

20.1 Procesos confiables

Los procesos software confiables son procesos que están adaptados a la prevención y detección de defectos. Los procesos confiables están bien definidos y son repetibles, e incluyen una variedad de actividades de verificación y validación. Un proceso bien definido es un proceso que ha sido estandarizado y documentado. Un proceso repetible es aquel que no depende de un juicio e interpretación individual. Independientemente de la gente implicada en el proceso, la organización puede estar segura de que el proceso se realizará con éxito. Se estudia la importancia de los procesos para conseguir calidad del software y la mejora de los procesos en el Capítulo 28. Las características fundamentales de los procesos confiables se muestran en la Figura 20.2.

Un proceso confiable debería incluir siempre actividades de verificación y validación bien planificadas y organizadas cuyo objetivo es asegurar el descubrimiento de defectos residuales en el software antes de que éste sea desplegado. Las actividades del proceso que se llevan a cabo para prevenir y detectar los defectos comprenden:

1. *Inspecciones de requerimientos.* Tal y como se explicó en el Capítulo 7, éstas pretenden descubrir problemas en la especificación del sistema. Una proporción elevada de defectos en el software entregado es consecuencia de errores en los requerimientos. Si éstos pueden descubrirse y eliminarse de la especificación, entonces esta clase de defectos se minimizarán.
2. *Gestión de requerimientos.* La gestión de requerimientos, tratada en el Capítulo 7, está relacionada con la realización de un seguimiento de los cambios en los requerimientos y extender dicho seguimiento a lo largo del diseño y la implementación. Muchos errores en los sistemas entregados son el resultado de no asegurar que realmente se incluya un cambio de requerimientos en el diseño e implementación del sistema.
3. *Verificación de modelos.* La verificación de modelos implica el uso de herramientas CASE para analizar los modelos del sistema y asegurar su consistencia interna y externa. La *consistencia interna* significa que un modelo del sistema es consistente; la *consistencia externa* significa que diferentes modelos del sistema (por ejemplo, un modelo de estados y un modelo de objetos) son consistentes.

Documentable	**El proceso debería tener un modelo de procesos definido que determine las actividades del proceso y la documentación que tiene que producirse durante estas actividades.**
Estandarizado	**Debería estar disponible un conjunto completo de estándares de desarrollo del software que definan cómo el software tiene que ser producido y documentado.**
Auditable	**El proceso debería ser comprensible para las personas ajenas a los participantes en el proceso, quienes pueden probar que los estándares del proceso se están siguiendo y realizar sugerencias para la mejora del proceso.**
Diverso	**El proceso debería incluir diversas actividades de verificación y validación.**
Robusto	**El proceso debería ser capaz de recuperarse de fallos de ejecución de actividades del proceso individual.**

Figura 20.2 Características de los procesos confiables.

4. *Inspecciones de diseño y código.* Tal y como indica en el Capítulo 22, las inspecciones de diseño y código a menudo se basan en listas de verificación de defectos comunes y pretenden descubrir y eliminar estos defectos antes de la prueba del sistema.
5. *Análisis estático.* El análisis estático es una técnica automática de análisis de programas en la que el programa se analiza con detalle para encontrar condiciones potencialmente erróneas. Esto se trata en el Capítulo 22.
6. *Planificación y gestión de las pruebas.* Debería diseñarse un conjunto completo de pruebas para el sistema y gestionar cuidadosamente el proceso de pruebas para asegurar una cobertura completa de las pruebas y el seguimiento de los requerimientos y el diseño del sistema a través de las pruebas. Las pruebas se estudian en el Capítulo 23.

Una posible fuente de error en sistemas críticos consiste en incluir el componente erróneo o la versión errónea de un componente en el sistema. Para evitar esto, se necesita utilizar una gestión de configuraciones estricta. La *gestión de configuraciones* está relacionada con la gestión de los cambios del software y con la realización de un seguimiento de las versiones de un sistema y sus componentes. Este tema se trata en el Capítulo 29.

20.2 Programación confiable

La programación confiable implica el uso de técnicas y construcciones de programación que contribuyen a prevenir y tolerar los defectos. Los defectos en los programas tienen lugar debido a que los programadores cometen errores. Mientras que algunos errores son debidos a malas interpretaciones de la especificación, otros tienen lugar debido a la complejidad de los programas o al uso de construcciones intrínsecamente propensas a error. Para alcanzar la confiabilidad, sin embargo, sería preciso realizar diseños simples, proteger la información de accesos no autorizados y minimizar el uso de construcciones de programación no seguras.

Las técnicas de programación para la prevención de defectos se basan en el hecho de que hay una distinción entre defectos y fallos de ejecución. Un *fallo de ejecución* es algo que es observable para los usuarios de un sistema, mientras que un *defecto* es una característica interna del sistema. Si un defecto se manifiesta en un programa en ejecución, es posible tolerarlo detectándolo y realizando la acción de recuperación antes de que se convierta en un fallo de ejecución del sistema. En esta sección, se analiza el uso de las construcciones de manejo de excepciones para hacer que los programas sean más tolerantes a defectos y más fáciles de comprender.

20.2.1 Información protegida

Un principio sobre protección que se adopta en organizaciones militares es el principio de la «necesidad de conocer». Sólo a aquellos individuos que necesitan conocer una parte concreta de la información para llevar a cabo su cometido se les proporciona dicha información. La información que no está directamente relacionada con su trabajo es desautorizada.

Cuando se programa, debería adoptarse un principio análogo para controlar el acceso a los datos del sistema. A los componentes del programa se les debería permitir el acceso solamente a los datos que necesitan para su implementación. Pueden protegerse otros datos mediante el uso de reglas de ámbito del lenguaje de programación para ocultarlos desde otras partes del programa. Si se oculta información, ésta no puede ser viciada por los com-

ponentes del programa que se supone que no la utilizan. Si la interfaz sigue siendo la misma, la representación de los datos puede cambiarse sin afectar a otros componentes en el sistema.

La protección de la información es mucho más sencilla en Java que en lenguajes más antiguos como C o Pascal. Debido a que estos lenguajes no tienen construcciones de encapsulación tales como clases de objetos, los detalles de la implementación de las estructuras de datos no pueden ser protegidos. Otras partes del programa pueden acceder a la estructura directamente, lo que produce efectos secundarios no esperados cuando se realizan cambios.

Generalmente es una buena práctica, cuando se programa en un lenguaje orientado a objetos, proporcionar métodos que acceden y actualizan los valores de los atributos en lugar de permitir que otros objetos accedan a estos atributos directamente. Esto significa que puede cambiarse la representación del atributo sin preocuparse de cómo otros objetos utilizan el atributo. Es particularmente importante utilizar esta aproximación para estructuras de datos y otros atributos complejos.

La construcción de definiciones de interfaces Java hace posible utilizar esta aproximación y es posible declarar la interfaz de un objeto sin hacer referencia a su implementación. Esto se ilustra en la Figura 20.3. Los usuarios de los objetos de tipo **Queue** pueden insertar objetos en la cola, eliminarlos de la cola y consultar el tamaño de la cola. Sin embargo, en la clase que implementa esta interfaz, la implementación real de la cola debería ocultarse declarando los atributos y métodos como particulares de esa clase de objetos. La separación de las interfaces y su implementación es una parte esencial de la ingeniería del software basada en componentes.

Otro tipo de protección de información se ilustra en la Figura 20.4. En situaciones en las que un conjunto de valores limitado puede asignarse a alguna variable, estos valores deberían declararse como constantes. Lenguajes tales como C++ soportan tipos enumerados, pero en Java esto debe implementarse asociando estas restricciones con la declaración de la clase.

Por ejemplo, consideremos un sistema de señalización, implementado en Java, que soporte luces rojas, ámbar y verdes. Debería definirse un tipo **Signal** que incluya declaraciones de constantes que reflejen estos colores. Por lo tanto, es posible hacer referencia a **Signal.red**, **Signal.green** y así sucesivamente. Esto evita asignaciones accidentales de valores incorrectos

```
interface Queue {

    public void put (Object o) ;
    public void remove (Object o) ;
    public int size () ;

} //Queue
```

Figura 20.3 Una especificación de una Cola utilizando una declaración de interfaz Java.

```
class Signal {

    static public final int red = 1 ;
    static public final int amber = 2 ;
    static public final int green = 3 ;

    public int sigState ;
}
```

Figura 20.4 Una declaración de una Señal en Java que oculta el tipo de representación.

a las variables de tipo **Signal**. Por lo tanto, se están protegiendo variables de tipo **Signal** de asignaciones incorrectas y, al mismo tiempo, ocultando la representación de rojo, ámbar y verde. Pueden cambiarse estos valores constantes más tarde sin tener que realizar ningún otro cambio en el programa.

20.2.2 Programación segura

Los defectos en los programas, y por lo tanto muchos fallos de ejecución de éstos, son normalmente consecuencia de un error humano. Los programadores cometen errores debido a que pierden la pista de todas las relaciones entre las variables de estado. Escriben sentencias de programas que provocan un comportamiento inesperado y cambios en el estado del sistema. Las personas siempre cometerán errores, pero fue evidente a finales de la década de los 60 que algunas aproximaciones de programación eran más propensas a error que otras.

Dijkstra (Dijkstra, 1968) reconoció que la sentencia goto o salto incondicional era una construcción de programación intrínsecamente propensa a error. Hace difícil localizar los cambios de estado. Esta observación condujo al desarrollo de la programación estructurada. La programación estructurada es la programación sin sentencias goto, utilizando solamente bucles while y sentencias if como construcciones de control y un diseño mediante una aproximación descendente. La programación estructurada fue un hito importante en el desarrollo de la ingeniería del software debido a que fue el primer paso hacia una aproximación disciplinada del desarrollo del software.

Otras construcciones de lenguajes de programación y técnicas de programación son también intrínsecamente propensas a error. Es menos probable introducir defectos en los programas si se evitan o, al menos, se usan lo menos posible. Las construcciones potencialmente propensas a error comprenden:

1. *Números en coma flotante*. Los números en coma flotante son intrínsecamente imprecisos. Éste es un problema particular cuando hay comparaciones debido a que la imprecisión en la representación puede llevar a comparaciones incorrectas. Por ejemplo, 3,00000000 a veces puede representarse como 2,99999999 y a veces como 3,00000001. Una comparación podría mostrar estos valores como distintos. Los números en coma fija, que son aquellos en los que un número se representa con un número determinado de posiciones decimales, son más seguros debido a que son posibles las comparaciones exactas.
2. *Punteros*. Los punteros son construcciones de bajo nivel que almacenan direcciones que hacen referencia directamente a zonas de memoria de la máquina. Los errores en su uso pueden ser devastadores, debido a que permiten *aliasing* (explicado más adelante en esta lista) y debido a que pueden hacer más difícil de implementar la comprobación de los límites de los vectores y de otras estructuras.
3. *Asignación dinámica de memoria*. La memoria del programa puede asignarse en tiempo de ejecución en lugar de en tiempo de compilación. El peligro de esto es que la memoria puede no liberarse de forma que eventualmente el sistema puede quedarse sin memoria disponible. Éste puede ser un error muy difícil de detectar debido a que el sistema puede ejecutarse con éxito durante un periodo de tiempo largo antes de que surja ningún problema.
4. *Paralelismo*. El paralelismo es peligroso debido a las dificultades de predecir los efectos sutiles de las interacciones temporales entre los procesos en paralelo. Los pro-

blemas temporales normalmente no pueden detectarse mediante la inspección de programas, y la peculiar combinación de circunstancias que ocasionan un problema temporal puede no darse durante la prueba del sistema. El paralelismo puede ser inevitable, pero su uso debería ser cuidadosamente controlado para minimizar las dependencias entre los procesos. Las facilidades de los lenguajes de programación tales como los hilos de ejecución (*threads*) de Java ayudan a gestionar el paralelismo para que puedan evitarse algunos errores de programación.

5. *Recursión.* La recursión se produce cuando un procedimiento o método se llama a sí mismo o llama a otro procedimiento, el cual a su vez llama al procedimiento original que realizó la llamada. Su uso puede generar programas concisos, pero puede ser difícil seguir la lógica de los programas recursivos. Por lo tanto, los errores de programación son más difíciles de detectar. Los errores de recursión pueden provocar la asignación de toda la memoria del sistema a medida que se crean las variables temporales de la pila.
6. *Interrupciones.* Son un modo de forzar la transferencia de control a una sección de código independientemente del código que se esté ejecutando actualmente. Los peligros de esto son evidentes: la interrupción puede provocar la terminación de una operación crítica.
7. *Herencia.* El problema de la herencia en la programación orientada a objetos es que el código asociado con un objeto no está todo en un mismo sitio. Esto hace más difícil comprender el comportamiento del objeto. Por lo tanto, es más probable que los errores de programación no se encuentren. Además, la herencia cuando se combina con el enlace dinámico puede provocar problemas temporales en tiempo de ejecución. En diferentes momentos, podrían llamarse a distintas instancias de un método específico y se consumirían tiempos diferentes buscando la instancia correcta del método.
8. *Aliasing.* Esto ocurre cuando se utiliza más de un nombre para referirse a la misma entidad en un programa. Es fácil para los lectores de los programas omitir sentencias que cambian la entidad cuando tienen que considerar varios nombres.
9. *Vectores no limitados.* En lenguajes tales como C, los vectores son simplemente formas de acceder a la memoria, y pueden hacerse asignaciones más allá del final de un vector. El soporte de ejecución del sistema no comprueba que las asignaciones realmente se refieren a los elementos del vector. Una vulnerabilidad conocida de la protección es el desbordamiento del búfer, en el que un intruso deliberadamente construye un programa para escribir en la memoria más allá del final de un búfer que se implementa como un vector.
10. *Procesamiento de entradas por defecto.* Algunos sistemas proporcionan un procesamiento de las entradas por defecto independientemente de la entrada presentada al sistema. Esto es un agujero en la protección que un intruso puede aprovechar para presentar al programa entradas no esperadas y que no sean rechazadas por el sistema.

Algunos estándares para el desarrollo de sistemas de seguridad críticos prohíben completamente el uso de estas construcciones. Sin embargo, esta posición extrema no es normalmente práctica. Todas estas construcciones y técnicas son útiles, pero tienen que utilizarse con cuidado. Siempre y cuando sea posible, sus potenciales efectos peligrosos deberían controlarse utilizándolos dentro de tipos abstractos de datos u objetos. Éstos actúan como «cortafuegos» limitando el daño ocasionado si se producen errores.

Los diseñadores de Java han reconocido algunos de los problemas de las construcciones propensas a error. El lenguaje no incluye sentencias goto, utiliza un recolector de basura de manera que no necesita asignar memoria de forma dinámica, y no soporta punteros o vectores no limitados. Sin embargo, la representación numérica de Java es tal que el soporte de ejecución del sistema no detecta desbordamiento de memoria, y todavía son posibles fallos de ejecución debidos a errores de operaciones en punto flotante.

20.2.3 Manejo de excepciones

Durante la ejecución del programa, los errores o eventos no esperados ocurren inevitablemente. Esto puede darse debido a un defecto en el programa o puede ser el resultado de circunstancias externas no predecibles. Un error o un evento inesperado que ocurra durante la ejecución de un programa se denomina una *excepción*. Las excepciones pueden ser provocadas por condiciones hardware o software. Ejemplos de excepciones podrían ser un fallo en el suministro de energía al sistema, un intento de acceso a datos no existentes, y desbordamiento de memoria numérico.

Cuando ocurre una excepción, ésta debe ser manejada por el sistema. Esto puede hacerse dentro del mismo programa o puede implicar la transferencia de control a un mecanismo de manejo de excepciones del sistema. Normalmente, el mecanismo de manejo de excepciones del sistema simplemente informa del error y abandona la ejecución. Por lo tanto, para asegurar que las excepciones del programa no provocan fallos de ejecución del sistema, debería definirse un manejador de excepciones para todas las posibles excepciones que puedan ocurrir y asegurarse de que todas las excepciones se manejan de forma explícita.

En lenguajes de programación tales como C, las sentencias if deben utilizarse para detectar la excepción y transferir el control al código de manejo de excepciones. Esto significa que se tienen que comprobar explícitamente las excepciones en cualquier sitio en el programa en donde puedan ocurrir. Sin embargo, esto añade complejidad y, por lo tanto, incrementa la probabilidad de que se cometan errores y de que la excepción no sea manejada correctamente.

Algunos lenguajes de programación, como Java, C++ y Ada, incluyen construcciones que soportan el manejo de excepciones de forma que no se necesitan sentencias condicionales adicionales para comprobar las excepciones. En su lugar, el lenguaje de programación incluye un tipo de construcción especial (a menudo denominado **Exception**), y diferentes excepciones pueden declararse con este tipo. Cuando ocurre una situación excepcional, la excepción es capturada y el soporte de ejecución del lenguaje transfiere el control a un manejador de excepciones. Éste es una sección de código que declara los nombres de las excepciones y las acciones adecuadas para manejar cada excepción.

En Java, pueden declararse nuevos tipos de excepciones extendiendo la clase **Exception**. Las excepciones son provocadas en Java utilizando una sentencia **throw**. El manejador de una excepción se indica por la palabra clave **catch**, seguida por un bloque de código que maneja la excepción.

La Figura 20.5 ilustra el uso de las excepciones en Java. Este ejemplo, parte del software para la bomba de insulina introducida en el Capítulo 3, es un controlador de un sensor que lee el valor de la glucosa en la sangre desde un sensor. La primera declaración en la Figura 20.5 muestra cómo se declaran las excepciones en Java. Se extiende la clase de objetos denominada **Exception**, y el constructor del método define el código a implementar cuando la excepción es provocada. En este caso, se activa una alarma.

```
class SensorFailureException extends Exception {

    SensorFailureException (String msg) {
        super (msg) ;
        Alarm.activate (msg) ;
    }
} // SensorFailureException

class Sensor {

    int readVal () throws SensorFailureException {

    try {
        int theValue = DeviceIO.readInteger () ;
        if (theValue < 0)
            throw new SensorFailureException ("Sensor failure") ;
        return theValue ;
    }
    catch (deviceIOException e)
        {
            throw new SensorFailureException (" Sensor read error ") ;
        }
    } // readVal
} // Sensor
```

Figura 20.5
Excepciones para manejar fallos de ejecución en la bomba de insulina.

La clase **Sensor** proporciona un único método denominado **readVal**, que incluye una sentencia throw en su declaración. Esto significa que una **SensorFailureException** puede ser capturada desde dentro del método, pero el método que hace la llamada se espera que proporcione un manejador para **SensorFailureException**. Normalmente es mejor manejar las excepciones en el método que hace la llamada debido a que este método conoce lo que se pretende hacer con el resultado del método al que se llama. Sin embargo, como se muestra más adelante, hay algunas situaciones en las que las excepciones se manejan localmente para asegurar que el resultado de una llamada a un método es siempre válido.

La palabra clave **try** indica que puede provocarse una excepción en el siguiente bloque de código. La excepción **SensorFailureException** se lanza si un valor menor que cero es devuelto cuando se comprueba el sensor. **DeviceIO.readInteger** puede lanzar una excepción denominada **deviceIOException**, por lo que a continuación de la palabra clave **catch** también debería incluirse un manejador para esta excepción. En este caso, el manejador simplemente provoca una excepción de fallo de ejecución de sensor para indicar que el objeto que realiza la llamada debería manejar la excepción.

El manejo de excepciones también puede utilizarse para simplificar los programas y hacerlos más fáciles de leer y entender. Esto reduce la probabilidad de error del programador e incrementa la posibilidad de que los inspectores de programas encuentren cualquier problema que exista. El manejo de excepciones se utiliza para separar el código de manejo de error del código que maneja el procesamiento normal. Por lo tanto, es posible leer y comprender cada una de estas secciones del código por separado.

Esto se ha ilustrado en la Figura 20.6. Esta clase Java es una implementación de un controlador de temperatura de un congelador de comida. La temperatura requerida puede fijarse entre −18 y −40 grados Celsius. La comida congelada puede comenzar a descongelarse y las bacterias se vuelven activas a temperaturas por encima de −18 grados. El sistema de control

```
class FreezerController {

    Sensor tempSensor = new Sensor () ;
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal () ;
    final float dangerTemp = (float) -18.0 ;
    final long coolingTime = (long) 200000.0 ;

    public void run ( ) throws InterruptedException {
    try {
        Pump.switchIt (Pump.on) ;
        do {
            if (freezerTemp > tempDial.setting ())
                if (Pump.status == Pump.off)
                {
                    Pump.switchIt (Pump.on) ;
                    Thread.sleep (coolingTime) ;
                }
            else
                    if (Pump.status == Pump.on)
                        Pump.switchIt (Pump.off) ;

            if (freezerTemp > dangerTemp)
                throw new FreezerTooHotException () ;
            freezerTemp = tempSensor.readVal () ;

        } while (true) ;

    } // try block
    catch (FreezerTooHotException f)
    { Alarm.activate ( ) ; }
    catch (InterruptedException e)
    {
        System.out.println ("Thread exception") ;
        throw new InterruptedException ( ) ;
    }

    } //run
} // FreezerController
```

Figura 20.6
Excepciones en un controlador de temperatura de un congelador.

mantiene esta temperatura alternando el encendido y apagado de una bomba refrigerante de acuerdo con el valor de un sensor de temperatura. Si la temperatura requerida no puede mantenerse, el controlador dispara una alarma.

En la implementación Java, la temperatura del congelador se obtiene interrogando a un objeto denominado **tempSensor**, y la temperatura requerida se obtiene inspeccionando un objeto denominado **tempDial**. Un objeto bomba (**Pump**) responde a las señales para cambiar su estado. Una vez que la bomba ha sido encendida, el sistema espera durante algún tiempo (llamando a **Thread.sleep**) para que la temperatura se reduzca. Si no ha disminuido suficientemente, se lanza una excepción denominada **FreezerTooHotException**.

El manejador de excepciones (situado al final del código) captura esta excepción y activa el objeto **Alarm**. También se incluye un manejador para la excepción **InterrupmtedException**, que puede ser provocada por **Thread.sleep**. Éste registra la excepción, y a continuación vuelve a provocar la excepción para su manejo en el método principal.

20.3 Tolerancia a defectos

Un sistema tolerante a defectos puede continuar en funcionamiento después de que se manifiesten algunos defectos en el programa. Los mecanismos de tolerancia a defectos en el sistema aseguran que estos defectos del sistema no provocan fallos de funcionamiento del sistema. Se necesita tolerancia a defectos en situaciones en las que un fallo de funcionamiento del sistema podría provocar un accidente catastrófico o en las que una pérdida de funcionamiento del sistema pudiese causar grandes pérdidas económicas. Por ejemplo, las computadoras de un avión deben estar en funcionamiento hasta que el avión aterrice; las computadoras en un sistema de tráfico aéreo deben estar disponibles continuamente mientras los aviones estén en el aire.

Existen cuatro aspectos a considerar en la tolerancia a defectos:

1. *Detección de defectos*. El sistema debe detectar un defecto que podría conducir a un fallo de ejecución del sistema. Generalmente, esto implica comprobar que el estado del sistema es consistente.
2. *Evaluación de los daños*. Se deben detectar las partes del estado del sistema que han sido afectadas por el defecto.
3. *Recuperación de defectos*. El sistema debe restaurar su estado a un estado «seguro» conocido. Esto puede conseguirse corrigiendo el estado dañado (recuperación de errores hacia adelante) o restaurando el sistema a un estado «seguro» conocido (recuperación de errores hacia atrás).
4. *Reparación de defectos*. Esto implica modificar el sistema para que no vuelva a aparecer el defecto. Sin embargo, muchos defectos del software se manifiestan como estados transitorios. Ello se debe a una combinación peculiar de entradas del sistema. No es necesario realizar ninguna reparación y el procesamiento normal puede continuarse inmediatamente después de la recuperación de los defectos.

Podría pensarse que las facilidades para la tolerancia a defectos son innecesarias en sistemas que han sido desarrollados utilizando técnicas que evitan la introducción de defectos. Si no hay defectos en el sistema, podría parecer que no hay ninguna posibilidad de un fallo de ejecución del sistema. Sin embargo, «libre de defectos» no significa «libre de fallos de ejecución». Tan sólo significa que el programa se corresponde con su especificación. La especificación puede contener errores u omisiones y puede estar basada en suposiciones incorrectas sobre el entorno del sistema. Y, desde luego, nosotros nunca podemos demostrar de forma concluyente que un sistema está completamente libre de defectos. En los sistemas que tienen los requerimientos más altos de fiabilidad y disponibilidad, se necesita utilizar redundancia y diversas aproximaciones de prevención de defectos y de tolerancia a defectos.

20.3.1 Detección de defectos y evaluación de daños

La primera etapa de la tolerancia a defectos es detectar que un defecto (un estado del sistema erróneo) o bien ha ocurrido u ocurrirá a menos que se realice inmediatamente alguna acción. Para hacer esto, se necesita conocer cuándo el valor de una variable de estado es ilegal o cuándo las relaciones entre variables de estado no se mantienen. Por lo tanto, se necesita definir restricciones de los estados que determinan las condiciones que deberían cumplirse para to-

dos los estados legales. Si estos predicados son falsos, entonces ha tenido lugar un defecto. Algunos ejemplos de restricciones de estados que se aplican al software de la bomba de insulina se muestran en la Figura 20.7. De forma deliberada no se han escrito estas restricciones como sentencias de aserciones Java por las razones que se explicarán más adelante.

Existen dos tipos de detección de defectos que se pueden utilizar:

1. *Detección de defectos preventiva.* En este caso, el mecanismo de detección de defectos se inicia antes de que se produzca un cambio en el estado. Si se detecta un estado potencialmente erróneo, entonces el cambio de estado no se realiza.
2. *Detección de defectos retrospectiva.* En este caso, el mecanismo de detección de defectos se inicia después de que el estado del sistema ha sido cambiado para comprobar si ha tenido lugar un defecto. Si se descubre un defecto, se provoca una excepción y se utiliza un mecanismo de reparación para recuperarse de ese defecto.

Puede utilizarse la detección de defectos preventiva cuando las restricciones de los estados que han sido definidas se aplican solamente a variables de estados individuales. Por ejemplo, usted puede utilizar esta aproximación cuando el valor de una variable de estado debe estar dentro de un rango definido. La detección de defectos preventiva evita la sobrecarga de la reparación de un defecto, ya que el estado del sistema siempre será válido, si bien no necesariamente correcto. Sin embargo, el sistema debe tener un mecanismo para continuar su funcionamiento ante la presencia de un estado incorrecto si se quiere evitar un fallo de ejecución del sistema.

En Java, la forma más segura de implementar la detección de defectos preventiva es comprobar de forma explícita la presencia de defectos y utilizar el mecanismo de manejo de excepciones en el lenguaje para indicar que un estado erróneo del sistema ha sido detectado. Esto se ilustra en la Figura 20.8. Ésta es una implementación de una clase en la que los valores de las instancias de la clase son restringidos a números positivos pares. Si se hace un intento de asignar un número que es impar o menor que cero, entonces se provoca una excepción.

En Java 1.4, se introdujo una facilidad para aserciones cuando las restricciones de los estados puedan ser definidas con una sentencia assert. Por lo tanto, para especificar que un número debería ser positivo y par, se debería escribir:

```
assert n >= 0 & n%2 == 0: «Value must be positive and even»
```

El soporte de ejecución del sistema comprueba que la condición se cumple y, si no es así, provoca un error y hace que se imprima el mensaje asociado.

Sin embargo, la facilidad de aserciones en Java fue diseñada realmente para lograr a descubrir inconsistencias de estados durante el desarrollo y depuración en lugar de soportar programación tolerante a defectos. Es posible activar y desactivar la comprobación de aserciones, de forma que usted no puede confiar en que las aserciones estén siempre activadas.

```
// The dose of insulin to be delivered must always be greater
// than zero and less that some defined maximum single dose

insulin_dose >= 0 & insulin_dose <= insulin_reservoir_contents

// The total amount of insulin delivered in a day must be less
// than or equal to a defined daily maximum dose

cumulative_dose <= maximum_daily_dose
```

Figura 20.7 Restricciones de los estados que se aplican en la bomba de insulina.

Además, no es posible asociar un tipo específico de excepción con cada aserción, por lo que no pueden identificarse fallos de funcionamiento individuales en las aserciones. Esto fue una decisión de diseño deliberada; los diseñadores no pretendían que fuese posible recuperarse de una acción después de que una aserción falle.

La detección de defectos preventiva es posible cuando se conoce el rango de valores que pueden asignarse a una variable de estado. Sin embargo, cuando un valor válido depende del valor asignado a otras variables en el estado, la detección de defectos preventiva puede ser imposible. Por ejemplo, supongamos que el programa lee tres valores, A, B y C, en ese orden. La restricción del estado es:

$A < B \ \& \ B < C$

No puede aplicarse la detección de defectos preventiva cuando se lea el valor de A debido a que no se conoce cuál será el valor de B y C. Del mismo modo, cuando se lea B, no se puede comprobar que es menor que C. Por lo tanto, necesita utilizar la detección de defectos retrospectiva, comprobando la restricción del estado después de que todos los valores hayan sido leídos. Si la restricción es falsa, entonces se puede realizar alguna acción para restaurar la consistencia del sistema.

Una forma de implementar la detección de defectos retrospectiva en Java consiste en asociar una función de comprobación con un objeto. Esta función puede llamarse después de que los cambios en el estado hayan tenido lugar para asegurar que se cumplen las restricciones del

```
class PositiveEvenInteger {

    int val = 0 ;

    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;

    }// PositiveEvenInteger

    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException ();
        else
            val = n ;
    } // assign

    int toInteger ()
    {
        return val ;
    } //to Integer

    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val) ;
    } // equals

} //PositiveEven
```

Figura 20.8 Clase PositiveEvenInteger en Java.

estado. Éstas pueden llamarse cuando sea necesario —puede que no se necesite comprobar el estado después de que cada cambio haya tenido lugar—. La siguiente interfaz puede ser utilizada para funciones de comprobación:

```
interface CheckableObject {
    public boolean check();
}
```

Los objetos a comprobar son instancias de una clase de objetos que implementa esta interfaz, por lo que cada objeto tiene una función de comprobación asociada. Cada clase de objetos implementa su propia función de comprobación que define las restricciones particulares que se aplican a los objetos de esa clase. Cuando se comprueba un estado en su totalidad, se utiliza enlace dinámico para asegurar que se aplica la función de comprobación adecuada para la clase del objeto que se está comprobando. Puede verse un ejemplo de esto en la Figura 20.9, en donde la función check comprueba que los elementos de un vector satisfacen alguna restricción.

La detección de defectos retrospectiva, que utiliza restricciones de estados que se aplican a más de una variable de estado, se ilustra en la Figura 20.10. En este ejemplo, la comprobación de detección de defectos se aplica a elementos consecutivos de un vector y comprueba que el vector esté ordenado.

```
class RobustArray {

    // Checks that all the objects in an array of objects
    // conform to some defined constraint

    boolean [] checkState ;
    CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray

    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
                {
                    checkState [i] = true ;
                    hasBeenDamaged = true ;
                }
            else
                checkState [i] = false ;
         }
         if (hasBeenDamaged)
              throw new ArrayDamagedException () ;
    } //assessDamage

} // RobustArray
```

Figura 20.9 Una clase vector con evaluación de daños.

La evaluación de daños implica analizar el estado del sistema para estimar el alcance de la corrupción del estado. La evaluación de daños es necesaria cuando no se puede evitar realizar un cambio de estado o cuando un defecto tiene lugar por una secuencia inválida de cambios de estado correctos individualmente.

El papel de los procedimientos de evaluación de daños no es recuperarse del defecto sino evaluar qué partes del espacio de estados han sido afectadas por el defecto. Los daños sólo pueden ser evaluados si es posible aplicar alguna «función de validación» que comprueba que el estado es inconsistente. Si se encuentran inconsistencias, éstas son resaltadas o indicadas de alguna manera.

La Figura 20.9 muestra una forma de implementar la evaluación de daños en Java. La estructura de datos denominada RobustArray es una colección de objetos de tipo CheckableObject. La clase que implementa el tipo CheckableObject debe incluir un método denominado check que pueda probar si el valor del objeto satisface alguna restricción. Este método de comprobación se asocia con este objeto en lugar de con el objeto RobustArray debido a que los detalles de la comprobación dependen del uso del tipo CheckableObject.

El método assessDamage en la clase RobustArray examina cada elemento del vector y comprueba que su estado es correcto. Si uno o más elementos del vector no satisfacen las restricciones del estado definidas en la función check, entonces los elementos dañados se registran en el vector checkState. A continuación se lanza una excepción denominada ArrayDamageException. Se debe incluir en el método que realiza la llamada un manejador para esta excepción que gestione el daño. Éste puede utilizar la información en checkState para decidir lo que hay que hacer.

```
class SafeSort {

    static void sort ( int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];

        // copy the input array

        for (int i = 0; i < intarray.length ; i++)
            copy [i] = intarray [i] ;
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
            if (order == Sort.ascending)
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError () ;
            else
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError () ;
         } // try block
        catch (SortError e )
        {
            for (int i = 0; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Array not sorted") ;
        } //catch
    } // sort
} // SafeSort
```

Figura 20.10
Procedimiento de ordenación seguro con recuperación de errores hacia atrás.

Otras técnicas de detección de defectos y de evaluación de daños dependen de la representación de los estados del sistema y del tipo de aplicación. Estas técnicas de evaluación de daños incluyen:

1. El uso de comprobaciones de código y sumas de verificación en las comunicaciones de datos y comprobaciones de dígitos en datos numéricos.
2. El uso de enlaces redundantes en estructuras de datos que contienen punteros.
3. El uso de temporizadores en sistemas concurrentes.

Las comprobaciones de código (Fujiwara y Pradhan, 1990) pueden utilizarse cuando los datos son intercambiados y en los que una suma de verificación se asocia con los datos numéricos. Una suma de verificación es un valor único que se calcula aplicando alguna función matemática a los datos. Esta suma de verificación es calculada por el emisor, que aplica la función de suma de verificación a los datos y añade el valor de esa función a los datos a transferir. El receptor aplica la misma función a los datos y compara el valor calculado con la suma de verificación añadida. Como las funciones son las mismas, si estos valores difieren, entonces el propio dato tiene que haber cambiado. Esta técnica puede utilizarse para detectar intrusiones en la protección así como corrupciones de los datos accidentales o deliberadas.

Cuando se utilizan estructuras enlazadas de datos, la representación puede hacerse redundante incluyendo referencias hacia atrás. Es decir, para cada referencia desde A hasta B, existe una referencia comparable desde B hasta A. También se puede contar el número de elementos en la estructura. La comprobación puede determinar si las referencias hacia adelante y hacia atrás son consistentes (éstas deberían referirse la una a la otra) y si el tamaño de la estructura calculado es el mismo.

Cuando los procesos deben reaccionar dentro de un periodo de tiempo específico, puede instalarse un temporizador de vigilancia. Un temporizador de vigilancia es un temporizador que debe reinicializarse por el proceso en ejecución cuando se completa su acción. Comienza al mismo tiempo que un proceso, y temporiza la ejecución del proceso. Un controlador puede consultarlo a intervalos regulares. Si, por alguna razón, el proceso falla en su terminación, el temporizador de vigilancia no se reinicializa. Por lo tanto, el controlador puede detectar que ha surgido un problema y realizar alguna acción para forzar la terminación del proceso.

20.3.2 Recuperación y reparación de defectos

La recuperación de defectos es el proceso de modificar el espacio de estados del sistema para que los efectos del defecto sean eliminados o reducidos. El sistema puede continuar funcionando, quizás de forma algo degradada. La **recuperación hacia adelante** implica intentar corregir el sistema dañado y crear el estado esperado. La **recuperación hacia atrás** restaura el estado del sistema a un estado «correcto» conocido.

La recuperación de errores hacia adelante sólo es posible en situaciones en las que la información del estado incluye redundancia. Existen dos situaciones generales (ambas descritas en la sección previa) en las que pueden aplicarse técnicas de recuperación de errores:

1. *Cuando los datos del código están dañados.* El uso de técnicas de codificación que añaden redundancia a los datos permite corregir los errores así como detectarlos.
2. *Cuando las estructuras enlazadas están dañadas.* Cuando los punteros hacia adelante y hacia atrás se incluyen en la estructura de datos, la estructura puede volverse a crear —si un número suficiente de punteros permanece sin dañar—. Esta técnica se utiliza frecuentemente para sistemas de ficheros y reparación de bases de datos.

La recuperación de errores hacia atrás es una técnica más simple que restaura el estado a un estado seguro conocido después de haberse detectado un error. La mayoría de los sistemas de bases de datos incluyen recuperación de errores hacia atrás. Cuando un usuario inicia un cálculo en la base de datos, se inicia una transacción. Los cambios realizados durante esa transacción no se incorporan inmediatamente en la base de datos. La base de datos sólo se actualiza después de que la transacción termine y no se haya detectado ningún problema. Si la transacción falla, la base de datos no se actualiza.

Las transacciones permiten la recuperación de errores puesto que no realizan cambios en la base de datos hasta que se han completado. Sin embargo, no permiten recuperación de cambios de estados que son válidos pero incorrectos. Los puntos de comprobación es una técnica que puede recuperarse de esta situación. El estado del sistema se duplica periódicamente. Cuando se descubre un problema, un estado correcto puede restaurarse a partir de una de estas copias.

Como ejemplo de cómo puede implementarse la recuperación hacia atrás utilizando puntos de comprobación, consideremos la clase Java **SafeSort** mostrada en la Figura 20.10 que incluye código para la detección de errores y recuperación hacia atrás.

El método crea un punto de comprobación copiando el vector antes de la operación de ordenación. En este ejemplo, se utiliza por simplicidad una ordenación de burbuja, pero obviamente puede utilizarse cualquier algoritmo de ordenación. Si hay un error en el algoritmo de ordenación y el vector no está correctamente ordenado, esto se detecta mediante comprobaciones explícitas del orden de los elementos en el vector. Si el vector no está correctamente ordenado, se provoca una excepción **SortException**. El manejador de excepciones no intenta reparar el problema, sino que restaura el valor original del vector y relanza **SortError** para indicar al método que realiza la llamada que la ordenación no ha tenido éxito. Es entonces responsabilidad del método que realiza la llamada el decidir cómo continuar la ejecución.

Como se ha sugerido antes, muchos defectos del software son transitorios, y no se requiere una reparación explícita para corregir las condiciones que provocan tales defectos. Éstos desaparecen en una ejecución posterior del sistema. En donde no se dé el caso, es posible llevar a cabo alguna acción reparadora. La acción reparadora del software más usual es reinicializar el sistema, reinicializando el estado a sus valores iniciales seguros (Huang y Kintala, 1993). A veces, esto puede realizarse sin parar el sistema si la inicialización es rápida y las peticiones de servicio pueden posponerse. Otras alternativas de reparación, como la reconfiguración dinámica, normalmente sólo son posibles cuando se ha hecho una provisión explícita desde el diseño del sistema.

20.4 Arquitecturas tolerantes a defectos

En muchos sistemas, es posible implementar la tolerancia a defectos del software incluyendo de forma explícita comprobaciones y acciones de recuperación en el software. Esto se denomina programación defensiva. Sin embargo, esta aproximación no puede tratar de forma efectiva los defectos del sistema que tienen lugar a partir de interacciones entre el hardware y el software. Además, un mal entendimiento de los requerimientos puede implicar que tanto el código del sistema como la defensa asociada sean incorrectos.

Para la mayoría de los sistemas críticos, en particular aquellos con requerimientos de disponibilidad restringidos, puede necesitarse una arquitectura específica del sistema diseñada para soportar la tolerancia a defectos. Ejemplos de sistemas que utilizan esta aproximación

de tolerancia a defectos son los sistemas en los aviones que deben estar en funcionamiento durante todo el vuelo, los sistemas de telecomunicaciones y los sistemas críticos de órdenes y control. Pullum (Pullum, 2001) describe diferentes tipos de arquitecturas tolerantes a defectos que han sido propuestas.

Durante muchos años ha habido una necesidad de construir hardware tolerante a defectos. La técnica más comúnmente utilizada de tolerancia a defectos hardware está basada en la noción de redundancia modular triple (TMR). La unidad hardware se reproduce tres veces (o algunas veces más). La salida de cada unidad se envía a un comparador de salidas que normalmente se implementa como un sistema de votaciones. Si una de las unidades falla y no produce la misma salida que el resto de las unidades, se pasa por alto su salida. Un gestor de defectos puede intentar reparar la unidad defectuosa de forma automática, pero si esto es imposible, el sistema se reconfigura automáticamente para dejar fuera de servicio esa unidad. Seguidamente el sistema continúa funcionando con dos unidades (Figura 20.11).

Esta aproximación a la tolerancia a defectos cubre la mayoría de los fallos de ejecución del hardware que son el resultado de fallos en los componentes en lugar de defectos de diseño. Por lo tanto, es probable que los componentes fallen de forma independiente. Se supone que, cuando son completamente funcionales, todas las unidades hardware funcionan de acuerdo con su especificación. Por lo tanto, hay una baja probabilidad de fallos simultáneos en los componentes en todas las unidades hardware.

Por supuesto, todos los componentes podrían tener un defecto de diseño y entonces todos producirían la misma respuesta (errónea). Utilizando unidades hardware que tienen una especificación común, pero que se han diseñado y construido por diferentes fabricantes, se reduce la probabilidad de un modo de fallo generalizado de este tipo. Se supone que la probabilidad de que diferentes grupos cometan el mismo error de fabricación o diseño es pequeña.

Si los requerimientos de disponibilidad y fiabilidad de un sistema son tales que se necesita utilizar hardware tolerante a defectos, entonces también puede necesitarse software tolerante a defectos. Existen dos aproximaciones relacionadas con la provisión de software tolerante a defectos (Figuras 20.12 y 20.13). Ambas técnicas han sido derivadas del modelo hardware en el que se incluyen los componentes redundantes (o quizás los sistemas redundantes) y los componentes defectuosos se dejan fuera de servicio.

Las dos aproximaciones para la tolerancia a defectos del software son:

1. *Programación con n-versiones.* Utilizando una especificación común, el sistema software se implementa en varias versiones por diversos equipos. Estas versiones se ejecutan en paralelo sobre computadoras diferentes. Sus salidas se comparan utilizando un sistema de votaciones y las salidas inconsistentes o las que no se producen a tiempo son rechazadas. Al menos deberían estar disponibles tres versiones del sistema para que dos versiones fuesen consistentes en el caso de que sólo una de ellas fallase. Ésta

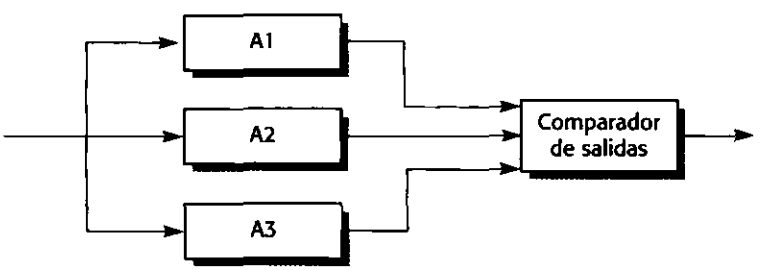


Figura 20.11 Redundancia modular triple para tratar los fallos de ejecución del hardware.

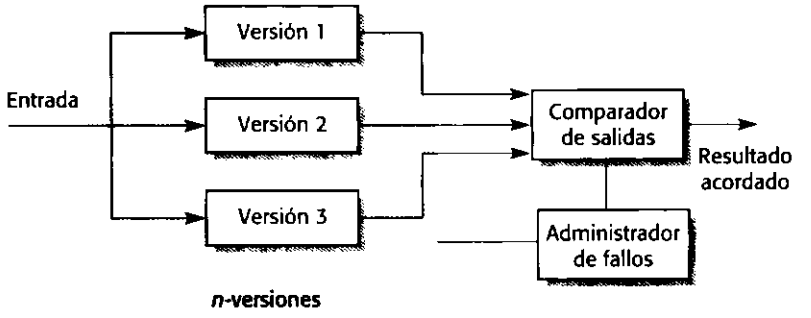


Figura 20.12 Programación con *n*-versiones.

es la aproximación usada más comúnmente para tolerancia a defectos del software. Ha sido utilizada en sistemas de señalización de vías ferroviarias, en sistemas de aviones y en sistemas de protección de reactores. Avizienis (Avizienis, 1985; Avizienis, 1995) describe esta aproximación.

2. *Bloques de recuperación.* En esta aproximación, cada componente del programa incluye una prueba para verificar que el componente se ha ejecutado con éxito. También incluye código alternativo que permite que el sistema vuelva hacia atrás y repita los cálculos si la prueba detecta un fallo de ejecución. De forma deliberada, las implementaciones son interpretaciones diferentes de la misma especificación. Se ejecutan en secuencia en lugar de en paralelo, de forma que no se requiere la replicación de hardware. En la programación con *n*-versiones, las implementaciones pueden ser distintas, pero no es infrecuente para dos o más equipos de desarrollo elegir los mismos algoritmos para implementar la especificación. Randell (Randell, 1975) y Randell y Xu (Randell y Xu, 1995) describen el método de bloques de recuperación.

La provisión de tolerancia a defectos del software requiere que el software sea ejecutado bajo el control de un controlador tolerante a defectos que asegure que se ejecuten los pasos relacionados con la tolerancia a defectos. Este controlador examina las salidas y las compara. Si difieren, se inician algunas acciones de recuperación. Laprie y otros (Laprie *et al.*, 1995) describen las arquitecturas de los sistemas tolerantes a defectos.

Ambas aproximaciones a la tolerancia a defectos hacen uso de la diversidad de diseño e implementación. Cuando se utilizan varias aproximaciones para implementar la misma especificación, es una suposición razonable que las diferentes versiones del software no incluyan

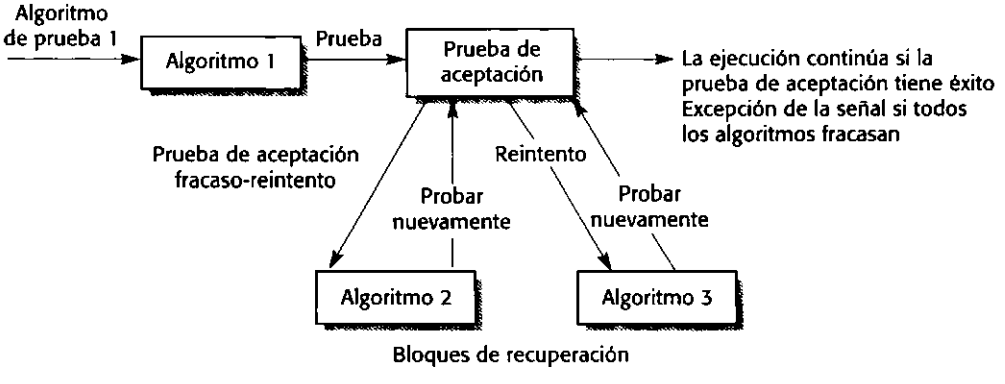


Figura 20.13 Bloques de recuperación.

los mismos defectos, por lo que los fallos comunes son poco probables. La diversidad se puede lograr de diferentes formas:

1. Incluyendo requerimientos que deberían usarse bajo diferentes aproximaciones de diseño. Por ejemplo, un equipo puede producir un diseño orientado a objetos y otro equipo puede producir un diseño orientado a funciones.
2. Exigiendo que las implementaciones fuesen escritas en diferentes lenguajes de programación. Por ejemplo, en un sistema con tres versiones, se podrían usar Ada, C++ y Java para escribir las versiones del software.
3. Exigiendo el uso de diferentes herramientas y entornos de desarrollo para el sistema.
4. Exigiendo de forma explícita diferentes algoritmos a utilizar en algunas partes de la implementación. Sin embargo, esto limita la libertad del equipo de diseño y puede ser difícil de reconciliar con los requerimientos de rendimiento del sistema.

Cada equipo de desarrollo debería trabajar con una especificación del sistema —la especificación V— que ha sido derivada de la especificación de los requerimientos del sistema (Avizienis, 1995). Así como se especifica la funcionalidad del sistema, la especificación V debería definir dónde se generan las comparaciones para las salidas del sistema. Los equipos de desarrollo para cada versión deberían trabajar de forma independiente para reducir la probabilidad de desarrollar malentendidos comunes sobre el sistema.

La diversidad del diseño ciertamente incrementa la fiabilidad total del sistema. Sin embargo, varios experimentos han sugerido que la suposición de que los equipos independientes de desarrollo no cometen los mismos errores no siempre es válida (Knight y Leveson, 1986; Brilliant *et al.*, 1990; Leveson, 1995). Los equipos de desarrollo pueden cometer los mismos errores debido a malas interpretaciones comunes de la especificación o debido a que de forma independiente llegan a los mismos algoritmos para resolver el problema. Los bloques de recuperación reducen la probabilidad de errores comunes debido a que se utilizan diferentes algoritmos de forma explícita para cada bloque de recuperación.

Las desventajas de ambas aproximaciones a la tolerancia a fallos es que están basadas en la suposición de que la especificación es correcta. Éstas no toleran errores en la especificación. En muchos casos, sin embargo, la especificación es incorrecta o incompleta, por lo que el sistema se comporta de forma inesperada. Una forma de reducir la posibilidad de errores comunes en la especificación es desarrollar las especificaciones V para el sistema de forma independiente y definir las especificaciones en distintos lenguajes. Un equipo de desarrollo podría trabajar a partir de una especificación formal, otro a partir de un modelo del sistema basado en estados, y el tercero a partir de una especificación en lenguaje natural. Esto ayuda a evitar algunos errores de la interpretación de las especificaciones, pero no evita el problema de errores en la especificación.

PUNTOS CLAVE

- **La confiabilidad en un programa se puede conseguir evitando la introducción de defectos, detectando y eliminando defectos antes del despliegue del sistema e incluyendo facilidades de tolerancia a defectos que permiten que el sistema permanezca en funcionamiento después de que un defecto ha provocado un fallo de ejecución del sistema.**

- El uso de redundancia y diversidad tanto en los procesos software como en los sistemas software es esencial para el desarrollo de sistemas confiables.
- El uso de un proceso repetible y bien definido es importante si se tienen que minimizar los defectos en un sistema. El proceso debería incluir actividades de verificación y validación en todas las etapas, desde la definición de los requerimientos hasta la implementación del sistema.
- Algunas técnicas y construcciones de programación, tales como sentencias goto, punteros, recursión, herencia y números en coma flotante, son intrínsecamente propensas a errores. Éstas no deberían utilizarse cuando se desarrollan sistemas confiables.
- Los cuatro aspectos de la programación de la tolerancia a defectos son detección de fallos de ejecución, evaluación de los daños, recuperación de defectos y reparación de defectos.
- La programación con *n*-versiones y bloques de recuperación son aproximaciones alternativas a las arquitecturas tolerantes a defectos en las que se mantienen copias redundantes del hardware y software. Ambas cuentan con la diversidad del diseño y el uso de un controlador tolerante a defectos para coordinar la ejecución de las unidades de programas redundantes.

LECTURAS ADICIONALES

Software Fault Tolerance Techniques and Implementation. Una descripción fácil de entender de técnicas para conseguir software tolerante a defectos y arquitecturas tolerantes a defectos. El libro también trata cuestiones generales de confiabilidad del software. (L. L. Pullum, 2001, Artech House.)

Handbook of Software Reliability Engineering. Esta colección incluye varios artículos que describen los bloques de recuperación y la programación con *n*-versiones. También incluye un buen artículo sobre arquitecturas de sistemas tolerantes a defectos. [M. R. Lyu (ed.), 1996, McGraw-Hill.]

EJERCICIOS

20.1 Dé cuatro razones de por qué casi nunca es rentable para las empresas asegurar que su software está libre de defectos.

20.2 Dé dos ejemplos de diversidad y actividades redundantes que podrían incorporarse en procesos confiables.

20.3 Explique por qué la herencia es una construcción potencialmente propensa a errores y por qué su uso debería minimizarse cuando se desarrollan sistemas críticos en un lenguaje orientado a objetos.

20.4 Comente los problemas de desarrollar y mantener sistemas «siempre activos» tales como el software para una central telefónica. ¿Cómo podrían utilizarse las excepciones en el desarrollo de tales sistemas?

20.5 Explique por qué debería manejar de forma explícita todas las excepciones en un sistema tolerante a defectos.

20.6 Describa brevemente las estrategias de recuperación de defectos hacia adelante y hacia atrás. ¿Por qué se usa más frecuentemente la recuperación de defectos hacia atrás que hacia adelante? Dé dos ejemplos de clases de sistemas en los que se podría utilizar la recuperación de errores hacia atrás.

20.7 ¿Qué es esencial para que la recuperación de errores hacia adelante pueda implementarse en un sistema tolerante a defectos? ¿Es posible la recuperación de errores hacia adelante en sistemas interactivos?

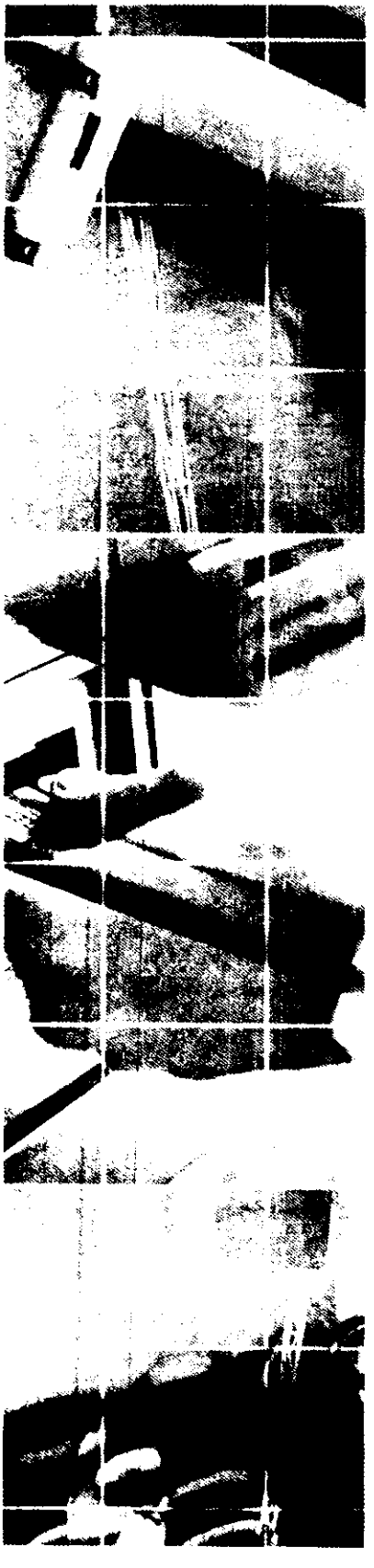
20.8 Diseñe un tipo abstracto de datos o clase de objetos denominado **RobustList** que implemente la recuperación de errores hacia adelante en una lista enlazada. Usted debería incluir operaciones para comprobar los daños en la lista y reconstruir la lista si ocurre algún daño. Suponga que puede comprobar los daños manteniendo referencias hacia adelante y hacia atrás desde y hasta miembros adyacentes de la lista.

20.9 Sugiera circunstancias en las que sea apropiado utilizar arquitecturas tolerantes a defectos cuando se implementa un sistema de control basado en software y explique por qué se requiere esta aproximación.

20.10 Se ha sugerido que el software de control para una máquina de terapia de radiaciones (utilizada para tratar a pacientes con cáncer) debería implementarse utilizando *n*-versiones. Comente si piensa que ésta es una buena sugerencia.

20.11 Dé dos razones de por qué las versiones de los sistemas en un sistema con *n*-versiones pueden fallar de forma similar.

20.12 El uso de las técnicas descritas aquí para producir software seguro obviamente implica costes adicionales considerables. ¿Qué costes adicionales pueden justificarse si 100 vidas pudiesen salvarse durante 15 años de la vida de un sistema? ¿Podrían justificarse los mismos costes si se salvasen 10 vidas? ¿Cuál es el valor de una vida humana? ¿La capacidad adquisitiva de la gente implicada hace que haya alguna diferencia en este juicio de valor?



21 Evolución del software

Objetivos

El objetivo de este capítulo es introducir la evolución del software para describir varias formas de modificar el software. Cuando haya leído este capítulo:

- comprenderá que el cambio es inevitable si los sistemas software tienen que seguir siendo útiles y que el desarrollo del software y la evolución del software pueden integrarse en un modelo en espiral;
- habrá aprendido sobre diferentes tipos de mantenimiento del software y los factores que afectan a los costes de mantenimiento;
- será consciente de los procesos implicados en la evolución del software, incluido el proceso de la reingeniería del software;
- comprenderá cómo los sistemas heredados pueden ser evaluados para decidir si deberían ser desechados, mantenidos, redesarrollados o reemplazados.

Contenidos

Después de que los sistemas hayan sido desarrollados, inevitablemente han de sufrir cambios si tienen que seguir siendo útiles. Una vez que el software comienza a utilizarse, surgen nuevos requerimientos y los requerimientos existentes cambian. Los cambios en los negocios a menudo generan nuevos requerimientos para el software existente. Algunas partes del software tienen que modificarse para corregir errores encontrados en su funcionamiento, adaptarlo a una nueva plataforma y mejorar su rendimiento u otras características no funcionales. El desarrollo del software, por lo tanto, no se detiene cuando un sistema es entregado, sino que continúa durante el tiempo de vida del sistema.

La evolución del software es importante debido a que las organizaciones actualmente son completamente dependientes de sus sistemas software y han invertido millones de dólares en ellos. Los sistemas software son activos de negocio críticos y las organizaciones deben invertir en los cambios del sistema para mantener el valor de estos activos. Por lo tanto, en grandes compañías, la mayor parte del presupuesto de software se dedica a mantener los sistemas existentes, y no debería sorprender que algunas figuras como las de Erlikh (Erlikh, 2000) sugieran que el 90% de los costes del software sean costes de evolución. Sin embargo, hay bastante incertidumbre en este porcentaje, ya que la gente quiere decir diferentes cosas cuando se refiere a la evolución o a los costes de mantenimiento.

Como se indica más adelante, los cambios posteriores al desarrollo no están relacionados simplemente con la reparación de defectos del software. La mayoría de los cambios son una consecuencia de que se generan nuevos requerimientos como respuesta a cambios en el negocio y en las necesidades de los usuarios. Como consecuencia, se puede pensar en la ingeniería del software como un proceso en espiral con requerimientos, diseño, implementación y pruebas que se realizan continuamente durante el tiempo de vida del sistema. Esto se ilustra en la Figura 21.1. Se comienza creando una Entrega 1 del sistema. Una vez entregada, los cambios propuestos y el desarrollo de la Entrega 2 comienzan casi inmediatamente. De hecho, la necesidad de evolución puede resultar obvia incluso antes de que se despliegue el sistema, de forma que posteriores entregas del software pueden estar en desarrollo antes de que la versión inicial haya sido entregada.

Éste es un modelo idealizado de la evolución del software que puede aplicarse en situaciones en las que una única organización es responsable tanto del desarrollo inicial del software como de la evolución del software. La mayoría de los productos software genéricos se desarrollan utilizando esta aproximación. Sin embargo, el software a medida puede desarro-

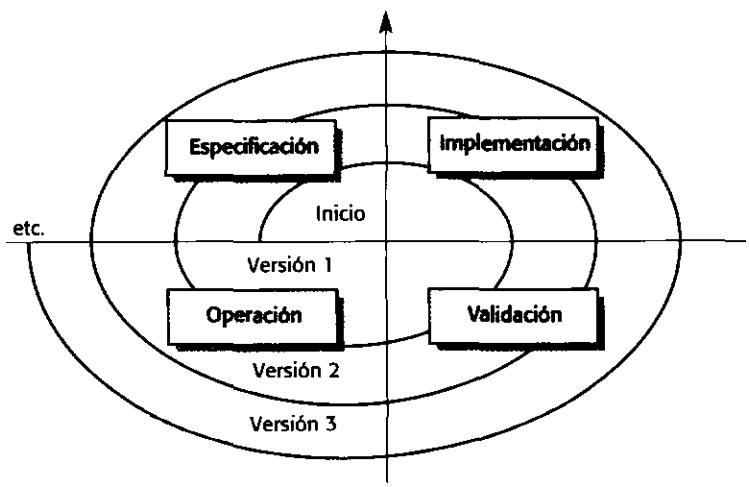


Figura 21.1
Un modelo en espiral del desarrollo y evolución.

llarse externamente, pero la evolución puede ser responsabilidad del personal de desarrollo del software del cliente. De forma alternativa, el usuario del software podría firmar un contrato distinto con una compañía externa para el soporte y la evolución del sistema.

En este caso, se producen a menudo discontinuidades en el proceso en espiral. Los documentos de requerimientos y diseño pueden no pasarse de una compañía a otra. Las compañías pueden combinar o reorganizar y heredar software de otras compañías, y entonces darse cuenta de que éste tiene que cambiarse. Cuando la transición desde el desarrollo a la evolución no es continua, el proceso de cambiar el software después de su entrega se denomina a menudo *mantenimiento del software*. Tal y como se expone más adelante en este capítulo, el mantenimiento implica actividades adicionales del proceso, como la comprensión del programa, además de las actividades normales del desarrollo del software.

21.1 Dinámica de evolución de los programas

La dinámica de evolución de los programas es el estudio de los cambios del sistema. La mayor parte del trabajo en esta área ha sido realizada por Lehman y Belady, primero en los años 70 y también en los 80 (Lehman y Belady, 1985). El trabajo continuó en los 90 cuando Lehman y otros investigaron la importancia de la realimentación en los procesos de evolución (Lehman, 1996; Lehman *et al.*, 1998; Lehman *et al.*, 2001). A partir de estos estudios, propusieron un conjunto de leyes (las leyes de Lehman) concernientes a los cambios de los sistemas. Señalan que estas leyes (hipótesis, realmente) son invariantes y ampliamente aplicables. Lehman y Belady examinaron el crecimiento y la evolución de varios sistemas software grandes. Las leyes propuestas, mostradas en la Figura 21.2, se derivaron de estas medidas.

Cambio continuado	Un programa que se usa en un entorno real necesariamente debe cambiar o se volverá progresivamente menos útil en ese entorno.
Complejidad creciente	A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja. Se deben dedicar recursos extras para preservar y simplificar la estructura.
Evolución prolongada del programa	La evolución de los programas es un proceso autorregulativo. Los atributos de los sistemas, tales como tamaño, tiempo entre entregas y el número de errores documentados, son aproximadamente invariantes para cada entrega del sistema.
Estabilidad organizacional	Durante el tiempo de vida de un programa, su velocidad de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.
Conservación de la familiaridad	Durante el tiempo de vida de un sistema, el cambio incremental en cada entrega es aproximadamente constante.
Crecimiento continuado	La funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción de los usuarios.
Decremento de la calidad	La calidad de los sistemas comenzará a disminuir a menos que dichos sistemas se adapten a los cambios en su entorno de funcionamiento.
Realimentación del sistema	Los procesos de evolución incorporan sistemas de realimentación multiagente y multibucle y éstos deben ser tratados como sistemas de realimentación para lograr una mejora significativa del producto.

Figura 21.2 Leyes de Lehman.

La primera ley establece que el mantenimiento de los sistemas es un proceso inevitable. A medida que el entorno del sistema cambia, surgen nuevos requerimientos y el sistema debe ser modificado. Cuando el sistema modificado vuelve a introducirse en el entorno, éste promueve más cambios en el entorno, de forma que el proceso de evolución se recicla.

La segunda ley establece que, a medida que se cambia un sistema, su estructura se degrada. La única forma de evitar este hecho, es invertir en mantenimiento preventivo en el que se consume tiempo mejorando la estructura del software sin añadir funcionalidades. Obviamente, esto implica costes adicionales, además de aquellos derivados de la implementación de los cambios requeridos del sistema.

La tercera ley es, quizás, la más interesante y discutida de las leyes de Lehman. Sugiere que los grandes sistemas tienen su propia dinámica que se establece en una etapa temprana en el proceso de desarrollo. Ésta determina las tendencias generales del proceso de mantenimiento del sistema y limita el número de posibles cambios en el sistema.

Lehman y Belady sugieren que esta ley es una consecuencia de factores estructurales que influencian y restringen los cambios de los sistemas, así como de factores organizacionales que afectan al proceso de evolución.

Una vez que un sistema excede algún tamaño mínimo, se vuelve más difícil de cambiar. Debido a que es más grande y más complejo, el sistema es más difícil de entender, y es más probable que los programadores cometan errores e introduzcan defectos en el sistema. Por lo tanto, la realización de pequeños cambios evita reducir la confiabilidad del sistema. Un gran cambio introducirá probablemente muchos defectos nuevos que limitarán el cambio útil entregado en la nueva versión del sistema.

Los grandes sistemas son producidos normalmente por grandes organizaciones, que tienen una burocracia interna que fija el presupuesto para el cambio de cada sistema y controla el proceso de toma de decisiones. Las organizaciones tienen que tomar decisiones sobre los riesgos y el valor de los cambios y los costes implicados. Tales decisiones llevan su tiempo. Durante ese tiempo, se pueden proponer otros cambios del sistema con mayor prioridad. Puede ser necesario retrasar los cambios originales hasta una fecha posterior. Por lo tanto, los procesos de toma de decisiones de la organización gobiernan la velocidad del cambio del sistema.

La cuarta ley de Lehman sugiere que la mayoría de los proyectos de programación grandes trabajan en lo que se denomina un estado *saturado*. Es decir, un cambio en los recursos o en el personal tiene efectos imperceptibles en la evolución a largo plazo del sistema. Esto concuerda con la tercera ley, que sugiere que la evolución del programa es independiente en gran medida de las decisiones de gestión. Esta ley confirma que los grandes grupos de desarrollo software son a menudo improductivos debido a que las sobrecargas en las comunicaciones dominan el trabajo del grupo.

La quinta ley de Lehman se refiere a los incrementos de los cambios en cada entrega del sistema. Añadir nueva funcionalidad a un sistema inevitablemente introduce nuevos defectos en el sistema. Cuanta más funcionalidad se añada en cada entrega, más defectos habrá. Por lo tanto, un incremento grande de funcionalidad en una entrega del sistema significa que tendrá que ir seguida de una entrega adicional en la que se reparen los nuevos defectos del sistema. Relativamente pocas nuevas funcionalidades se incluirán en esta entrega. La ley sugiere que no deberían presupuestarse incrementos grandes de funcionalidad en cada entrega sin tener en cuenta la necesidad de reparación de defectos.

Las cinco primeras leyes fueron las propuestas iniciales de Lehman; las restantes leyes fueron añadidas después de posteriores trabajos. Las leyes sexta y séptima son similares y dicen

esencialmente que los usuarios del software estarán cada vez más descontentos con dicho software a menos que éste se mantenga y se le añadan nuevas funcionalidades. La última ley refleja el trabajo más reciente sobre procesos de realimentación, si bien todavía no está claro cómo puede aplicarse en el desarrollo práctico del software.

Las observaciones de Lehman parecen razonables por lo general. Deberían tenerse en cuenta cuando se planifica el proceso de mantenimiento. Puede ser que las consideraciones del negocio requieran prescindir de ellas en algún momento. Por ejemplo, por razones comerciales, puede ser necesario realizar cambios mayores en el sistema en una única entrega. Las consecuencias probables de esto son que se requieran una o más entregas dedicadas a la reparación de los errores introducidos.

Puede parecer que las diferencias radicales que son obvias entre entregas de productos de programas violan las leyes de Lehman. Por ejemplo, Microsoft Word se transformó desde un sencillo procesador de textos que funcionaba con 256 K de memoria a un gigantesco sistema con multitud de características. Actualmente necesita muchos megabytes de memoria y un procesador rápido para que funcione. Su evolución parece contradecir las leyes de Lehman cuarta y quinta. Sin embargo, se presume que este programa no es realmente una secuencia de revisiones a partir de un programa núcleo común. Más bien, el nombre se ha conservado por razones comerciales, pero el programa en sí mismo ha sido reescrito y reestructurado más de una vez desde que fue entregado originalmente.

21.2 Mantenimiento del software

El mantenimiento del software es el proceso general de cambiar un sistema después de que éste ha sido entregado. El término se aplica normalmente a software a medida en donde grupos de desarrollo distintos están implicados antes y después de la entrega. Los cambios realizados al software pueden ser cambios sencillos para corregir errores de código, cambios más extensos para corregir errores de diseño o mejoras significativas para corregir errores de especificación o acomodar nuevos requerimientos. Los cambios se implementan modificando los componentes del sistema existente y añadiendo nuevos componentes al sistema donde sea necesario.

Existen tres tipos diferentes de mantenimiento de software:

1. *Mantenimiento para reparar defectos del software.* Por lo general, los errores de código son relativamente baratos de corregir; los errores de diseño son mucho más caros ya que implican reescribir varios componentes de los programas. Los errores de requerimientos son los más caros de reparar debido a que puede ser necesario un rediseño extenso del sistema.
2. *Mantenimiento para adaptar el software a diferentes entornos operativos.* Este tipo de mantenimiento se requiere cuando cambia algún aspecto del entorno del sistema, como por ejemplo el hardware, la plataforma del sistema operativo u otro software de soporte. El sistema de aplicaciones debe modificarse para adaptarse a estos cambios en el entorno.
3. *Mantenimiento para añadir o modificar las funcionalidades del sistema.* Este tipo de mantenimiento es necesario cuando los requerimientos del sistema cambian como respuesta a cambios organizacionales o del negocio. La escala de los cambios requeridos en el software es a menudo mucho mayor que en los otros tipos de mantenimiento.

En la práctica, no existe una clara distinción entre estos tipos de mantenimiento. Cuando se adapta el sistema a un nuevo entorno, pueden añadirse funcionalidades para aprovechar las ventajas de las nuevas características del entorno. Los defectos del software a menudo se muestran debido a que los usuarios utilizan el sistema en formas no anticipadas. La mejor forma de reparar estos defectos consiste en realizar cambios en el sistema que se adapten a su forma de trabajo.

Normalmente se reconocen estos tipos de mantenimiento, pero varias personas les dan distintos nombres. El *mantenimiento correctivo* se utiliza generalmente para referirse al mantenimiento para reparación de defectos. Sin embargo, el *mantenimiento adaptativo* algunas veces significa adaptarse a un nuevo entorno y puede significar adaptar el software a nuevos requerimientos. El *mantenimiento perfectivo* puede significar perfeccionar el software implementando nuevos requerimientos; en otros casos significa mantener la funcionalidad del sistema, pero mejorando su estructura y su rendimiento. Debido a esta incertidumbre en los nombres, se ha evitado usar todos estos términos en este capítulo.

Los trabajos de Lientz y Swanson (Lientz y Swanson, 1980) y los de Nosek y Palvia (Nosek y Palvia, 1990) sugieren que alrededor del 65% del mantenimiento está relacionado con la implementación de nuevos requerimientos, el 18% con cambios en el sistema para adaptarlo a un nuevo entorno operativo y el 17% para corregir defectos en el sistema (Figura 21.3). Para sistemas a medida, la distribución de estos costes todavía es aproximadamente correcta. La cuestión importante no es la de los porcentajes específicos, sino el hecho de que la reparación de defectos en los sistemas no es la actividad de mantenimiento más cara. La evolución del sistema para adaptarse a nuevos entornos y requerimientos nuevos o cambios en los mismos consume la mayor parte del esfuerzo de mantenimiento.

Los costes de mantenimiento constituyen una proporción de los costes de desarrollo y varían de un dominio de aplicación a otro. Guimaraes (Guimaraes, 1983) sugiere que los costes de mantenimiento para sistemas de aplicaciones corporativas son en general comparables con los costes de desarrollo de los sistemas. Para sistemas de tiempo real embebidos, los costes de mantenimiento pueden ser hasta cuatro veces mayores que los costes de desarrollo. Los requerimientos de alta fiabilidad y rendimiento de estos sistemas a menudo implican que los módulos tienen que estar estrechamente enlazados y, por lo tanto, serán difíciles de cambiar.

Normalmente resulta rentable invertir esfuerzo en el diseño e implementación de un sistema para reducir los costes de mantenimiento. Añadir nuevas funcionalidades después de la entrega es caro debido a que hay que emplear tiempo en la compresión del sistema y en el aná-

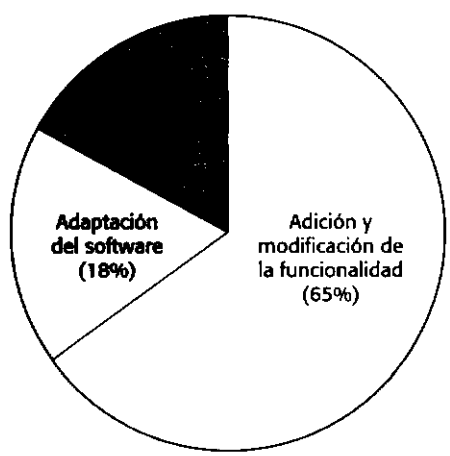


Figura 21.3 Distribución del esfuerzo de mantenimiento.

lisis del impacto de los cambios propuestos. Por lo tanto, el trabajo realizado durante el desarrollo para hacer que el software sea más fácil de entender y de cambiar, probablemente reduzca los costes de mantenimiento. Buenas técnicas de ingeniería del software tales como una especificación precisa, el uso de desarrollo orientado a objetos y gestión de configuraciones contribuyen a la reducción de los costes de mantenimiento.

La Figura 21.4 muestra cómo los costes totales del tiempo de vida pueden decrecer cuanto más esfuerzo se emplee durante el desarrollo del sistema para producir un sistema mantenible. Debido a la reducción potencial en los costes de comprensión, análisis y pruebas, existe un efecto multiplicador significativo cuando el sistema se desarrolla pensando en la mantenibilidad. Para el Sistema 1, se invierten unos costes de desarrollo extras de 25.000 dólares para hacer que el sistema sea más mantenible. Esto se convierte en un ahorro de 100.000 dólares en los costes de mantenimiento durante la vida del sistema, lo que supone un incremento del porcentaje en los costes de desarrollo que conduce a un porcentaje decreciente comparable en los costes totales del sistema.

Una razón importante de por qué los costes de mantenimiento son altos es que es más caro añadir funcionalidades después de que el sistema esté en funcionamiento que implementar la misma funcionalidad durante el desarrollo. Los factores clave que distinguen el desarrollo y el mantenimiento, y que conducen a costes de mantenimiento más elevados, son:

1. *Estabilidad del equipo.* Después de entregar un sistema, es normal que el equipo de desarrollo de disuelva y la gente trabaje en nuevos proyectos. El nuevo equipo o los individuos responsables del mantenimiento del sistema no comprenden dicho sistema o las razones de fondo de las decisiones de su diseño. Se dedica demasiado esfuerzo durante el proceso de mantenimiento en comprender el sistema antes de implementar cambios sobre él.
2. *Responsabilidad contractual.* El contrato para mantener un sistema normalmente está separado del contrato para desarrollar el sistema. El contrato de mantenimiento puede darse con una compañía diferente en lugar de con el desarrollador original del sistema. Este factor, junto con la ausencia de estabilidad del equipo, implica que no existe incentivo para que un equipo de desarrollo escriba el software para que sea fácil de cambiar. Si el equipo de desarrollo realiza ciertas acciones para ahorrar esfuerzo durante el desarrollo, lo hará sin preocuparse demasiado aunque esto signifique incrementar los costes de mantenimiento.
3. *Habilidades del personal.* El personal de mantenimiento a menudo no tiene experiencia y no está familiarizado con el dominio de la aplicación. El mantenimiento tiene una pobre imagen entre los ingenieros software. Está visto como un proceso que requiere menos habilidades que el desarrollo del sistema y a menudo se asigna al personal prin-

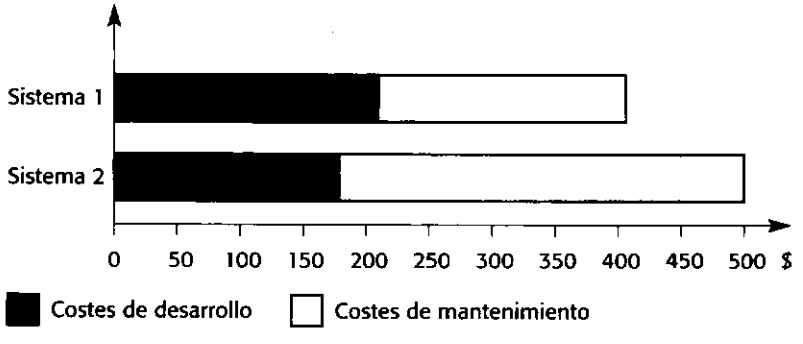


Figura 21.4
Costes de desarrollo y mantenimiento.

cipiante. Además, los sistemas antiguos pueden haberse escrito en lenguajes de programación obsoletos. El personal de mantenimiento puede no tener mucha experiencia de desarrollo en estos lenguajes y debe aprenderlos para mantener el sistema.

4. *Edad y estructura del programa.* A medida que pasa el tiempo, la estructura de los programas tiende a degradarse con los cambios, por lo que se vuelve más difícil de comprender y modificar. Algunos sistemas han sido desarrollados sin técnicas modernas de ingeniería del software. Pueden no haber sido nunca bien estructurados y quizá estén optimizados para su eficiencia en lugar de para su comprensibilidad. La documentación del sistema puede haberse perdido o ser inconsistente. Los sistemas antiguos pueden no haber sido sometidos a gestión de configuraciones, por lo que a menudo se emplea mucho tiempo en encontrar las versiones correctas de los componentes del sistema a cambiar.

Los tres primeros problemas surgen del hecho de que muchas organizaciones todavía consideran el desarrollo y el mantenimiento como actividades independientes. El mantenimiento se ve como una actividad de segunda categoría, y no es un incentivo emplear más dinero durante el desarrollo para reducir los costes de los cambios en el sistema. La única solución a largo plazo a este problema es aceptar que los sistemas raramente tienen un tiempo de vida definido, pero continuarán en uso, de alguna forma, durante un periodo de tiempo indefinido. Tal y como se sugirió en la introducción, debería pensarse que los sistemas evolucionan durante su ciclo de vida a través de un proceso de desarrollo continuado.

El cuarto problema, el de la degradación de la estructura del sistema, es en cierto modo el más fácil de solucionar. Las técnicas de reingeniería del software (brevemente descritas más adelante en este capítulo) pueden aplicarse para mejorar la estructura del sistema y su comprensibilidad. Las transformaciones arquitectónicas pueden adaptar el sistema a un nuevo hardware. El trabajo de mantenimiento preventivo (esencialmente la reingeniería incremental) puede soportarse para mejorar el sistema y hacer que sea más fácil de cambiar.

21.2.1 Predicción del mantenimiento

Los gestores odian las sorpresas, especialmente si éstas conducen a costes elevados inesperados. Por lo tanto, se debería intentar predecir qué cambios del sistema son probables y qué partes del sistema son probablemente las más difíciles de mantener. También se debería intentar estimar los costes totales de mantenimiento para un sistema durante un periodo de tiempo determinado. La Figura 21.5 ilustra estas predicciones y cuestiones asociadas. Como es obvio, estas predicciones están estrechamente relacionadas:

1. La aceptación o no de un cambio en el sistema depende, hasta cierto punto, de la mantenibilidad de los componentes del sistema afectados por dicho cambio.
2. La implementación de los cambios del sistema tiende a degradar la estructura de dicho sistema y, por lo tanto, reduce su mantenibilidad.
3. Los costes de mantenimiento dependen del número de cambios, y los costes de la implementación de los cambios dependen de la mantenibilidad de los componentes del sistema.

La predicción del número de peticiones de cambios para un sistema requiere entender la relación entre el sistema y su entorno. Algunos sistemas tienen una relación muy compleja con su entorno y los cambios en ese entorno inevitablemente provocan cambios en el sistema. Para evaluar las relaciones entre un sistema y su entorno, debería tenerse en cuenta:

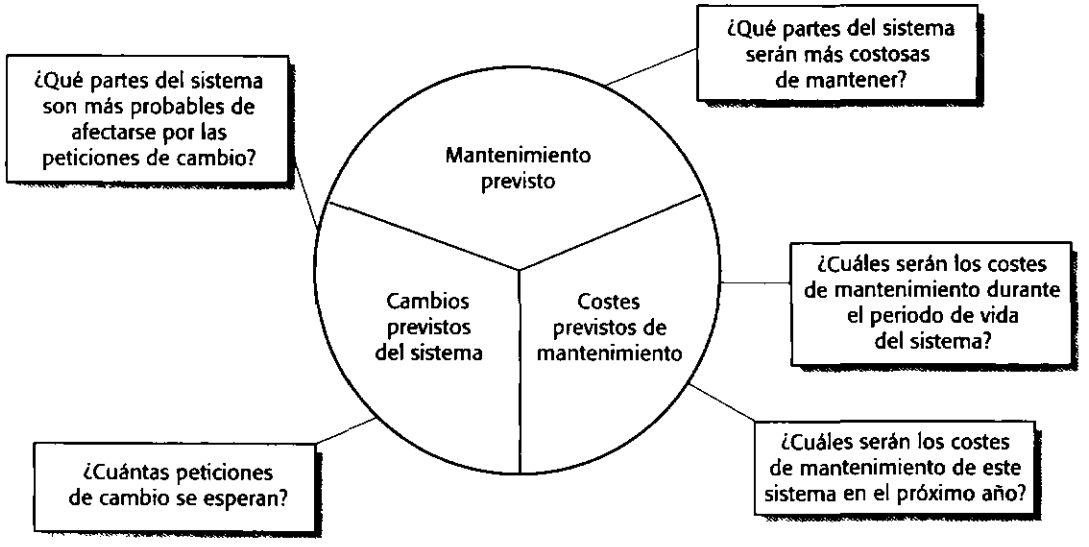


Figura 21.5 Predicción del mantenimiento.

1. *El número y la complejidad de las interfaces del sistema.* Cuanto mayor sea el número de interfaces y más complejas sean éstas, es más probable que se hagan más peticiones de cambio.
2. *El número de requerimientos del sistema intrínsecamente volátiles.* Tal y como indica en el Capítulo 7, los requerimientos que reflejan políticas y procedimientos organizacionales son probablemente más volátiles que los requerimientos que se basan en características estables del dominio.
3. *Los procesos de negocios en los que se utiliza el sistema.* Puesto que los procesos de negocios evolucionan, generan peticiones de cambio del sistema. Cuanto más procesos de negocios utilice el sistema, habrá más peticiones de cambio del sistema.

Para predecir la mantenibilidad del sistema, es necesario comprender el número y los tipos de relaciones entre los componentes del sistema. Se han realizado varios estudios sobre los diferentes tipos de complejidad en un sistema (McCabe, 1976; Halstead, 1977) y sobre las relaciones entre la complejidad y la mantenibilidad (Kafura y Reddy, 1987; Banker *et al.*, 1993). No es sorprendente que estos estudios señalaran que cuanto más complejo es un componente o sistema, más caro es de mantener.

Las medidas de la complejidad han resultado ser particularmente útiles para identificar componentes individuales de programas que probablemente sean especialmente caros de mantener. Kafura y Reddy (Kafura y Reddy, 1987) examinaron varios componentes de sistemas y comprobaron que el esfuerzo de mantenimiento tendía a centrarse en un pequeño número de componentes complejos. Sugirieron que, para reducir los costes de mantenimiento, se deberían reemplazar componentes del sistema particularmente complejos con alternativas más sencillas.

Después de que el sistema se haya puesto en funcionamiento, se pueden usar los datos del proceso para ayudar a predecir la mantenibilidad. Ejemplos de métricas del proceso que pueden utilizarse para evaluar la mantenibilidad son:

1. *El número de peticiones de mantenimiento correctivo.* Un crecimiento en el número de informes de fallos de ejecución puede indicar que se han introducido más errores

en el programa de los que se han corregido durante el proceso de mantenimiento. Esto puede suponer una disminución de la mantenibilidad.

2. *El tiempo medio requerido para el análisis de impacto.* Éste refleja el número de componentes del programa que se ven afectados por una petición de cambio. Si este tiempo se incrementa, implica que más y más componentes se ven afectados y que la mantenibilidad disminuye.
3. *El tiempo medio empleado en implementar una petición de cambio.* Éste no es el mismo que el tiempo para el análisis de impacto, aunque está relacionado con él. Se trata de la cantidad de tiempo que se necesita para modificar realmente el sistema y su documentación, después de que haya evaluado qué componentes se ven afectados. Un incremento en el tiempo necesario para implementar un cambio puede indicar una disminución en la mantenibilidad.
4. *El número de peticiones de cambio pendientes.* Un incremento en este número a lo largo del tiempo puede implicar una disminución en la mantenibilidad.

Para predecir los costes de mantenimiento se utiliza la información de predicciones sobre peticiones de cambio y sobre la mantenibilidad del sistema. La mayoría de los gestores combinan esta información con intuición y experiencia para estimar los costes. El modelo COCOMO 2 de estimación de costes (Boehm *et al.*, 2000), descrito en el Capítulo 26, sugiere que una estimación del esfuerzo de mantenimiento del software puede basarse en el esfuerzo de comprender el código existente y el esfuerzo de desarrollar nuevo código.

21.3 Procesos de evolución

Los procesos de evolución del software varían considerablemente dependiendo del tipo de software a mantener, los procesos de desarrollo utilizados en una organización y el personal implicado en el proceso. En algunas organizaciones, la evolución puede ser un proceso informal en el que la mayor parte de las peticiones de cambios surgen de conversaciones entre los usuarios del sistema y los desarrolladores. En otras compañías, hay un proceso formalizado en el que se produce documentación estructurada en cada etapa del proceso.

Las propuestas de cambios del sistema son los conductores de la evolución de los sistemas en todas las organizaciones. Estas propuestas de cambio pueden ser requerimientos existentes que no han sido implementados en el sistema entregado, peticiones para nuevos requerimientos y reparaciones de errores por parte de los stakeholders del sistema, y nuevas ideas y propuestas para mejoras en el software por parte del equipo de desarrollo del sistema. Tal y como se ilustra en la Figura 21.6, los procesos de identificación de cambios y evolución del sistema son cíclicos y continúan durante toda la vida del sistema.

Los procesos de evolución incluyen las actividades fundamentales de análisis de cambios, planificación de entregas, implementación del sistema y entrega de un sistema a los clientes. Se evalúa el coste e impacto de estos cambios para ver en qué medida se ve afectado el sistema por el cambio y cuánto cuesta implementar el cambio. Si los cambios propuestos son aceptados, se planifica una nueva entrega del sistema. Durante la planificación de entregas, se consideran todos los cambios propuestos (reparación de defectos, adaptación y nuevas funcionalidades). A continuación, se decide con qué lenguajes se va a implementar la siguiente versión del sistema. Se implementan y validan los cambios, y se entrega una nueva versión del sistema. El proceso entonces itera con un nuevo conjunto de

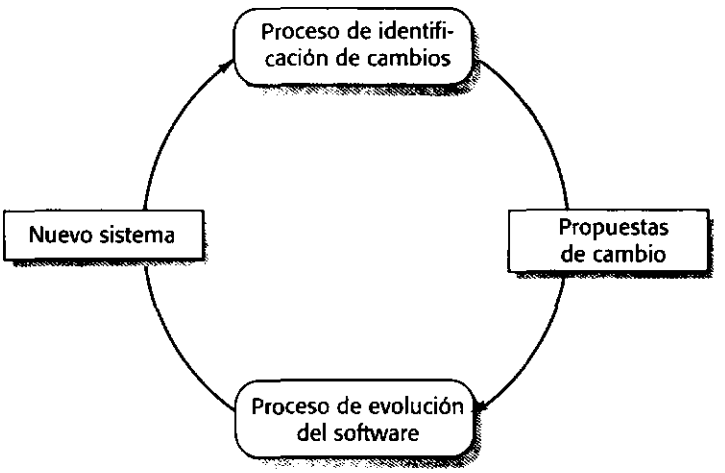


Figura 21.6 Identificación de los cambios y procesos de evolución.

cambios propuestos para la siguiente entrega. La Figura 21.7, adaptada de Arthur (Arthur, 1988), muestra un esquema de este proceso.

El proceso de implementación de los cambios es, esencialmente, una iteración del proceso de desarrollo en la que se diseñan, implementan y prueban las revisiones del sistema. Sin embargo, una diferencia fundamental es que la etapa inicial de la implementación de los cambios es la comprensión del programa. Durante esta fase, se tiene que entender cómo está estructurado el programa y cómo realiza su funcionalidad. Cuando se implementa un cambio, se usa este entendimiento para estar seguro de que el cambio implementado no afecta de forma adversa al sistema existente.

Idealmente, la etapa de implementación de los cambios de este proceso debería modificar la especificación del sistema, diseño e implementación para reflejar los cambios del sistema (Figura 21.8). Se proponen, analizan y validan nuevos requerimientos que reflejan los cambios del sistema. Los componentes del sistema son rediseñados e implementados y el sistema se vuelve a probar. Si es conveniente, pueden realizarse prototipos de los cambios propuestos como parte del proceso de análisis de los cambios.

A medida que se cambia el software, se desarrollan entregas sucesivas del sistema. Éstas están compuestas a partir de versiones de los componentes del sistema. Tiene que hacerse un seguimiento de estas versiones para asegurarse de que se utilizan las versiones correctas de los componentes en cada entrega del sistema. La gestión de configuraciones se trata en el Capítulo 29.

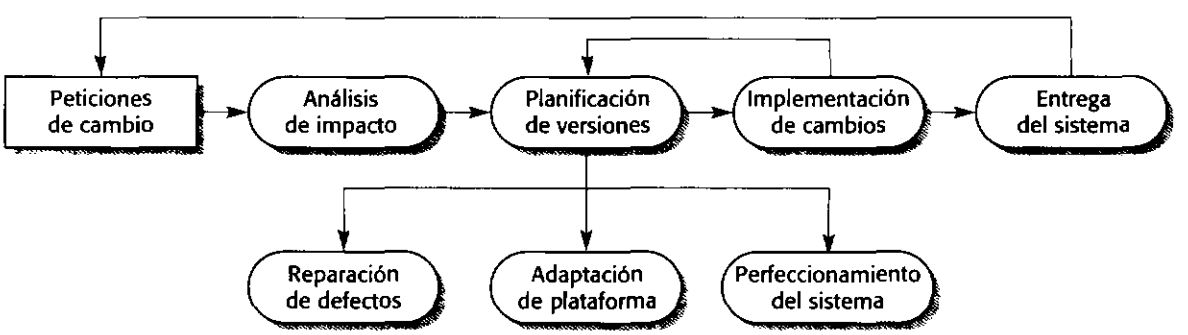


Figura 21.7 El proceso de evolución del sistema.

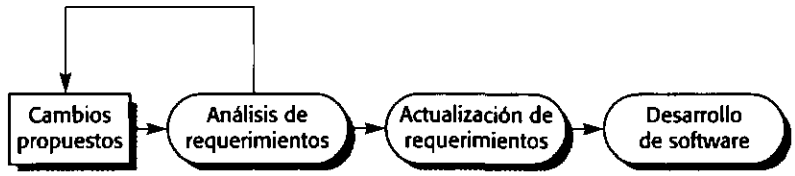


Figura 21.8 Implementación de los cambios.

Durante los procesos de evolución, los requerimientos se analizan con detalle y frecuentemente surgen las implicaciones de los cambios que no eran evidentes en la etapa más temprana del proceso de análisis de los cambios. Esto significa que los cambios propuestos pueden modificarse y pueden requerirse discusiones adicionales con el cliente antes de que sean implementados.

Algunas veces, las peticiones de cambio están relacionadas con problemas en el sistema que tienen que solucionarse de manera muy urgente. Estos cambios urgentes pueden tener lugar por tres razones:

1. Si ocurre un defecto serio en el sistema que tenga que ser reparado para permitir la continuación del funcionamiento normal.
2. Si los cambios en el entorno del sistema operativo tienen efectos inesperados que impiden el funcionamiento normal.
3. Si hay cambios no anticipados en las empresas que utilizan el sistema, tales como la aparición de nuevos competidores o la introducción de una nueva legislación.

En estos casos, la necesidad de realizar el cambio rápidamente significa que usted puede resultar imposible seguir el proceso de análisis formal del cambio. En lugar de modificar los requerimientos y diseño, se realiza una reparación de emergencia en el programa para resolver el problema inmediato (Figura 21.9). Sin embargo, el peligro está en que los requerimientos, el diseño del software y el código gradualmente se vuelven inconsistentes. Mientras se intenta documentar los cambios en los requerimientos y el diseño, pueden ser necesarias reparaciones de emergencia adicionales. Éstas tienen prioridad sobre la documentación. Finalmente, el cambio original se olvida y la documentación del sistema y el código nunca vuelven a ser consistentes.

Un problema adicional con las reparaciones de emergencia del sistema es que tienen que completarse lo más rápidamente posible. Se elige una solución funcional rápida en lugar de la mejor solución que concierne a la estructura del sistema. Esto acelera el proceso del envejecimiento del software de forma que futuros cambios son progresivamente más difíciles y los costes de mantenimiento se incrementan.

Idealmente, cuando se realizan reparaciones de código de emergencia, la petición de cambio debería seguir existiendo después de que los defectos hayan sido corregidos. Entonces ésta puede ser reimplementada más cuidadosamente después de un posterior análisis. Por supuesto, puede reutilizarse el código de la reparación. Otra solución mejor al problema puede descubrirse cuando se tiene más tiempo para realizar el análisis. Sin embargo, en la práctica, es casi siempre inevitable que estos cambios tengan una prioridad baja y, después de que se hayan hecho cambios adicionales en el sistema, no es realista volver a hacer las reparaciones de emergencia.

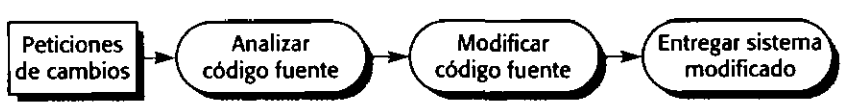


Figura 21.9 Proceso de reparaciones de emergencia

21.3.1 Reingeniería de sistemas

Tal y como se indica en la sección anterior, el proceso de evolución de los sistemas implica comprender el programa que tiene que cambiarse, y a continuación implementar estos cambios. Sin embargo, muchos sistemas, especialmente los sistemas heredados más antiguos (descritos en el Capítulo 2) son difíciles de comprender y de cambiar. Los programas pueden haber sido optimizados originalmente para su rendimiento o utilización de la memoria a expensas de su comprensibilidad o, a lo largo del tiempo, la estructura inicial del programa puede haberse corrompido por una serie de cambios.

Para simplificar los problemas de cambiar sus sistemas heredados, una compañía puede decidir hacer reingeniería sobre esos sistemas para mejorar su estructura y comprensibilidad. La reingeniería del software se refiere a la reimplementación de los sistemas heredados para hacerlos más mantenibles. La reingeniería puede implicar redocumentar el sistema, organizar y reestructurar el sistema, traducir el sistema a un lenguaje de programación más moderno, y modificar y actualizar la estructura y valores de los datos del sistema. La funcionalidad del software no se cambia y, normalmente, la arquitectura del sistema también sigue siendo la misma.

Hacer reingeniería de un sistema software tiene dos ventajas clave sobre aproximaciones más radicales a la evolución del sistema:

1. *Riesgo reducido.* Existe un alto riesgo en volver a desarrollar software crítico para los negocios. Pueden cometerse errores en la especificación, o puede haber problemas en el desarrollo. Los retrasos en la introducción del nuevo software pueden significar pérdidas en el negocio e incurrir en costes adicionales. Por ejemplo, en 1999 una gran compañía de comida en Estados Unidos tuvo retrasos en la introducción de un nuevo sistema de pedidos, lo que condujo a retrasos en las entregas de productos valoradas en 100 millones de dólares en una estación de máxima venta.
2. *Coste reducido.* El coste de hacer reingeniería es significativamente menor que el coste de desarrollar nuevo software. Ulrich (Ulrich, 1990) cita un ejemplo de un sistema comercial en el que los costes de reimplementación se estimaron en 50 millones de dólares. Al sistema se le aplicó reingeniería con éxito por 12 millones de dólares. Se presume que, con la tecnología moderna del software, el coste relativo de la reimplementación probablemente sea menor, pero aun así supera de forma considerable los costes de la reingeniería.

La distinción crítica entre reingeniería y nuevo desarrollo software es el punto de partida del desarrollo. En lugar de empezar con una especificación escrita, el sistema antiguo actúa como una especificación para el nuevo sistema. Chikofsky y Cross (Chikofsky y Cross, 1990) denominan al desarrollo convencional *ingeniería hacia adelante* para distinguirla de la reingeniería del software. Esta distinción se muestra en la Figura 21.10. La ingeniería hacia ade-

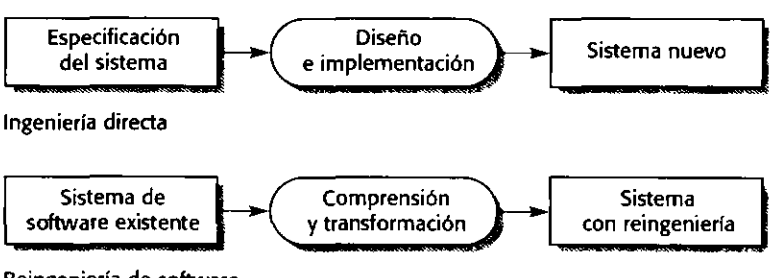


Figura 21.10 Ingeniería hacia adelante y reingeniería.

lante comienza con una especificación del sistema e implica el diseño e implementación de un nuevo sistema. La reingeniería comienza con un sistema existente y el proceso de desarrollo para su reemplazo se basa en comprender y transformar el sistema original.

La Figura 21.11 ilustra el proceso de reingeniería. La entrada del proceso es un programa heredado y la salida es una versión modularizada y estructurada del mismo programa. Durante la reingeniería del programa, los datos del sistema también sufren reingeniería. Las actividades de este proceso de reingeniería son:

1. *Traducción del código fuente*. El programa es convertido desde un lenguaje de programación antiguo a una versión más moderna del mismo lenguaje o a un lenguaje diferente.
2. *Ingeniería inversa*. El programa se analiza y se extrae información a partir de él. Esto ayuda a documentar su organización y funcionalidad.
3. *Mejora de la estructura de los programas*. La estructura de control del programa se analiza y modifica para hacerla más fácil de leer y comprender.
4. *Modularización de los programas*. Se agrupan las partes relacionadas del programa y se elimina la redundancia en donde resulta adecuado. En algunos casos, esta etapa puede implicar una transformación arquitectónica en la que un sistema centralizado pensado para una única computadora se modifica para ejecutarse sobre una plataforma distribuida.
5. *Reingeniería de datos*. Los datos procesados por el programa se cambian para reflejar los cambios en él.

La reingeniería del sistema no requiere necesariamente todos los pasos de la Figura 21.11. La traducción de código fuente puede no ser necesaria si el lenguaje de programación utilizado para desarrollar el sistema todavía está soportado por el suministrador del compilador. Si la reingeniería se realiza completamente con herramientas automáticas, entonces recuperar la documentación mediante ingeniería inversa puede no ser necesario. La reingeniería de datos sólo se requiere si las estructuras de datos del programa cambian durante la reingeniería del sistema. Sin embargo, la reingeniería del software siempre implica alguna reconstrucción del programa. Para hacer que el sistema que ha sufrido reingeniería interopere con el nuevo software, tienen que desarrollarse adaptadores de componentes, como se explicó en el Capítulo 19. Éste oculta las

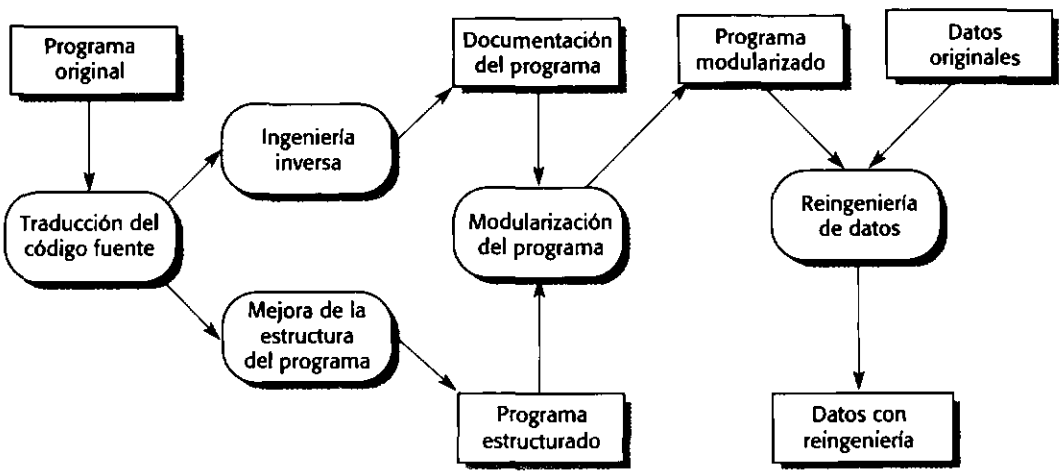


Figura 21.11 El proceso de reingeniería.

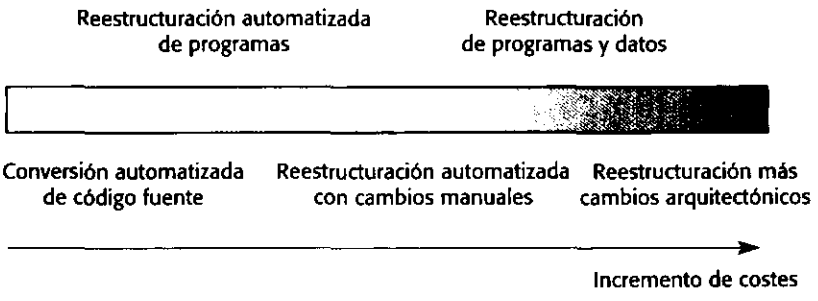


Figura 21.12 Aproximaciones de reingeniería.

interfaces originales del sistema software y presenta nuevas interfaces mejor estructuradas que pueden ser utilizadas por otros componentes. El proceso de envolver este sistema heredado es una técnica importante para desarrollar componentes reutilizables a gran escala.

Los costes de la reingeniería obviamente dependen de la magnitud del trabajo que tiene que llevarse a cabo, tal y como muestra la Figura 21.12. Los costes se incrementan desde la izquierda hacia la derecha para que la traducción de código fuente sea la opción más económica. La reingeniería, como parte de una migración arquitectónica, es la opción más cara.

Además de la amplitud de la reingeniería, los principales factores que afectan a los costes de reingeniería son:

1. *La calidad del software sobre el que se va a hacer reingeniería.* Cuanto más baja sea la calidad del software y su documentación asociada (si la hay), más altos serán los costes de reingeniería.
2. *Las herramientas de soporte disponibles para la reingeniería.* Normalmente no es rentable hacer reingeniería sobre un sistema software a menos que puedan utilizarse herramientas CASE para automatizar la mayor parte de los cambios en los programas.
3. *La amplitud de la conversión de datos requerida.* Si el sistema sobre el que se va a hacer reingeniería requiere que se conviertan grandes volúmenes de datos, el coste del proceso se incrementa de forma significativa.
4. *La disponibilidad de personal experto.* Si el personal responsable de mantener el sistema no puede implicarse en el proceso de reingeniería, los costes se incrementarán debido a que los ingenieros encargados de la reingeniería tienen que invertir una gran cantidad de tiempo en comprender el sistema.

La principal desventaja de la reingeniería del software es que existen límites prácticos a la extensión del sistema que puede ser mejorada mediante reingeniería. No es posible, por ejemplo, convertir un sistema diseñado utilizando una aproximación funcional en un sistema orientado a objetos. Los cambios arquitectónicos mayores o la reorganización radical de la gestión de datos del sistema no pueden realizarse de forma automática, por lo que se incurrirá en costes adicionales elevados. Aunque la reingeniería puede mejorar la mantenibilidad, el sistema al que se va a aplicar reingeniería probablemente no será tan mantenible como un nuevo sistema desarrollado utilizando métodos modernos de ingeniería del software.

21.4 Evolución de sistemas heredados

Para los sistemas software nuevos desarrollados utilizando procesos de ingeniería del software modernos tales como desarrollo iterativo y CBSE, es posible planificar cómo integrar el desarrollo del sistema y la evolución. Más y más compañías han empezado a comprender

que el proceso de desarrollo de los sistemas es un proceso del ciclo de vida en su totalidad y que una separación artificial entre el desarrollo del software y su mantenimiento no resulta útil. Sin embargo, existen todavía muchos sistemas heredados que son sistemas de negocio críticos. Éstos tienen que extenderse y adaptarse para las prácticas cambiantes de comercio electrónico.

Las organizaciones que cuentan con un presupuesto limitado para mantener y actualizar sus sistemas heredados tienen que decidir cómo obtener los mejores beneficios a su inversión. Esto significa que tienen que realizar evaluaciones realistas de sus sistemas heredados y a continuación decidir cuál es la estrategia más adecuada para la evolución de estos sistemas. Existen cuatro opciones estratégicas:

1. *Desechar completamente el sistema.* Esta opción debería elegirse cuando el sistema no constituye una contribución efectiva para los procesos de negocio. Esto ocurre cuando los procesos de negocio han cambiado desde que se instaló el sistema y ya no son completamente dependientes de éste. Esta situación es más común cuando las terminales mainframe fueron reemplazadas por PCs, y el software comercial sobre estas máquinas fue adaptado para proporcionar la mayor parte del soporte que el proceso de negocio necesita.
2. *Dejar el sistema sin cambios y continuar con un mantenimiento regular.* Esta opción debería elegirse cuando el sistema todavía es necesario pero es muy estable y los usuarios del sistema solicitan un número relativamente pequeño de peticiones de cambio.
3. *Hacer reingeniería del sistema para mejorar su mantenibilidad.* Esta opción debería elegirse cuando la calidad del sistema se ha degradado por los cambios continuos y cuando dichos cambios todavía son necesarios. Tal y como se ha indicado, este proceso puede incluir el desarrollo de nuevos componentes de interfaz para que el sistema original pueda trabajar con otros sistemas más nuevos.
4. *Reemplazar todo o parte del sistema con un nuevo sistema.* Esta opción debería elegirse cuando otros factores, como un nuevo hardware, implican que el sistema antiguo no puede continuar en funcionamiento o cuando los sistemas comerciales deberían permitir desarrollar el nuevo sistema con un coste razonable. En muchos casos puede adoptarse una estrategia de reemplazo evolutiva en la que los mayores componentes del sistema son reemplazados por sistemas comerciales con otros componentes reutilizados siempre que sea posible.

Naturalmente, estas opciones no son exclusivas, por lo que, cuando un sistema se compone de varios programas, pueden aplicarse diferentes opciones a distintas partes del sistema.

Cuando se está evaluando un sistema heredado, éste tiene que verse desde una perspectiva de negocio y desde una perspectiva técnica (Warren, 1998). Desde una perspectiva de negocio, tiene que decidirse si el negocio realmente necesita del sistema. Desde una perspectiva técnica, tiene que evaluarse la calidad de la aplicación software y del software y hardware de soporte del sistema. A continuación, debe utilizarse una combinación del valor del negocio y de la calidad del sistema para informar de lo que se ha decidido hacer con el sistema heredado.

Para ilustrar esto, supongamos que una organización tiene diez sistemas heredados. El valor del negocio y de la calidad de cada uno de estos sistemas se evalúa y se compara con otros creando un gráfico que muestra el valor relativo del negocio y de la calidad del sistema. Esto se ilustra en la Figura 21.13.

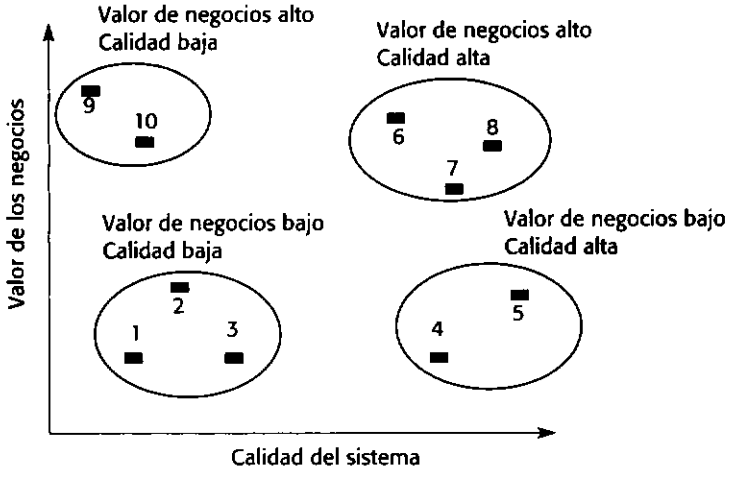


Figura 21.13 Evaluación de sistemas heredados.

A partir de la Figura 21.13, puede verse que existen cuatro clases de sistemas:

1. *Baja calidad y bajo valor de negocio.* Mantener estos sistemas en funcionamiento será caro y la tasa de beneficios para el negocio será bastante pequeña. Estos sistemas deberían desecharse.
2. *Baja calidad y alto valor de negocio.* Estos sistemas constituyen una importante contribución al negocio, por lo que no pueden desecharse. Sin embargo, su baja calidad significa que son caros de mantener. A estos sistemas debería aplicárseles reingeniería para mejorar su calidad o deberían ser reemplazados, si está disponible un sistema comercial adecuado.
3. *Alta calidad y bajo valor de negocio.* Éstos son sistemas que no contribuyen mucho al negocio, pero que no son muy caros de mantener. No vale la pena reemplazar estos sistemas para que su mantenimiento normal pueda continuar durante más tiempo, ya que no se requieren cambios caros y el hardware del sistema es operacional. Si es necesario realizar cambios caros, éstos deberían desecharse.
4. *Alta calidad y alto valor de negocio.* Estos sistemas tienen que seguir manteniendo su funcionamiento, pero su elevada calidad significa que no se tiene que invertir en su transformación o reemplazo de los sistemas. El mantenimiento normal de los sistemas debería continuar.

Para evaluar el valor de negocio de un sistema, se tiene que identificar a los *stakeholders* del sistema, tales como usuarios finales del sistema y sus gestores, y plantearles una serie de cuestiones sobre dicho sistema. Existen cuatro cuestiones básicas que deberían abordarse:

1. *El uso del sistema.* Si los sistemas sólo se utilizan de forma ocasional por un pequeño número de personas, pueden tener un bajo valor de negocio. Un sistema heredado puede haber sido desarrollado para satisfacer una necesidad de negocio que o bien ha cambiado o no puede satisfacerse de manera efectiva de otras formas.
2. *Los procesos de negocio que están soportados.* Cuando se introduce un sistema, pueden diseñarse procesos de negocio para explotar ese sistema. Sin embargo, cambiar estos procesos puede resultar imposible debido a que el sistema heredado no puede ser adaptado. Por lo tanto, un sistema puede tener un bajo valor de negocio debido a que no pueden introducirse nuevos procesos.

3. *Confiabilidad del sistema.* La confiabilidad del sistema no sólo es un problema técnico sino también un problema de negocio. Si un sistema no es confiable y los problemas afectan directamente a los clientes del negocio o suponen que las personas del negocio son apartadas de otras tareas para resolver estos problemas, el sistema tiene un bajo valor de negocio.
4. *Salidas del sistema.* La clave fundamental aquí es la importancia de las salidas del sistema para un funcionamiento con éxito del negocio. Si el negocio depende de estas salidas, entonces el sistema tiene un alto valor de negocio. A la inversa, si estas salidas pueden generarse fácilmente de alguna otra forma o si el sistema produce salidas que se usan raramente, entonces su valor de negocio puede ser bajo.

Por ejemplo, supongamos que una compañía proporciona un sistema de pedidos de viajes en que el personal responsable de organizar el viaje puede emitir órdenes con un agente de viajes concertado. A continuación, se entregan los billetes y se pasan las facturas a la compañía. Sin embargo, una evaluación del valor de negocio puede revelar que este sistema sólo es utilizado por un pequeño porcentaje de peticiones de viaje emitidas. Las personas que organizan los viajes pueden encontrar más barato y más conveniente tratar directamente con los proveedores de los viajes a través de sus sitios web. Este sistema todavía puede utilizarse, pero no hay un motivo real para mantenerlo. La misma funcionalidad está disponible desde sistemas externos.

De forma inversa, supongamos que una compañía ha desarrollado un sistema que realiza un seguimiento de todas las peticiones previas de los clientes, y automáticamente genera avisos a los clientes para volver a pedir los productos. Esto provoca un gran número de pedidos repetidos y mantiene a los clientes satisfechos debido a que ellos sienten que su proveedor está preocupado por sus necesidades. Las salidas de un sistema como éste son muy importantes para el negocio; por lo tanto, el sistema tiene un alto valor de negocio.

Para evaluar el software del sistema desde una perspectiva técnica, se necesita considerar tanto la aplicación del sistema en sí misma, como el entorno en el cual opera. El entorno incluye el hardware y todo el software de soporte asociado, como compiladores y enlazadores, necesarios para mantener el sistema. El entorno es importante debido a que muchos cambios en el sistema se obtienen a partir de cambios en el entorno, como actualizaciones del hardware o del sistema operativo.

En el proceso de evaluación del entorno, si es posible, deberían hacerse mediciones del sistema y de sus procesos de mantenimiento. Ejemplos de datos que pueden ser útiles son los costes de mantener el hardware del sistema y el software de soporte, el número de defectos hardware que se manifiestan durante algún periodo de tiempo y la frecuencia de las reparaciones aplicadas al software de soporte del sistema.

Los factores que deberían considerarse durante la evaluación del entorno se muestran en la Figura 21.14. Téngase en cuenta que estos factores no son todos características técnicas del entorno; también tiene que considerar la fiabilidad de los proveedores del hardware y del sistema de soporte. Si estos proveedores ya no están en el negocio, puede no haber soporte de mantenimiento para sus sistemas.

Para evaluar la calidad técnica de un sistema de aplicaciones, tiene que evaluarse una serie de rangos (Figura 21.15) que están relacionados fundamentalmente con la confiabilidad del sistema, las dificultades de mantener el sistema y la documentación del mismo. También tienen que recogerse datos cuantitativos del sistema que ayuden a juzgar su calidad. Ejemplos de datos cuantitativos que podrían obtenerse son:

1. *El número de peticiones de cambio del sistema.* Los cambios del sistema tienden a corromper la estructura del mismo y hacer que cambios adicionales sean más difíciles. Cuanto más alto sea este valor, más baja será la calidad del sistema.

Estabilidad del proveedor	**¿Existe todavía el proveedor? ¿Es éste financieramente estable y probablemente continúe existiendo? Si el proveedor ya no está en el negocio, ¿algún otro mantendrá los sistemas?**
Tasa de fallos de ejecución	**¿Tiene el hardware una tasa elevada de fallos de ejecución? ¿Falla el software de soporte y fuerza la reinicialización del sistema?**
Edad	**¿Es muy antiguo el hardware y el software? Cuanto más antiguo sea el hardware y el software de soporte, más obsoleto será. Puede todavía funcionar correctamente pero podría representar beneficios económicos significativos en el negocio si se cambiara por sistemas más modernos.**
Rendimiento	**¿Es adecuado el rendimiento del sistema? ¿Tienen los problemas de rendimiento un efecto significativo sobre los usuarios del sistema?**
Requerimientos de soporte	**¿Qué soporte local es requerido por el hardware y el software? Si existen costes elevados asociados con este soporte, puede merecer la pena considerar la sustitución del sistema.**
Costes de mantenimiento	**¿Cuáles son los costes de mantenimiento del hardware y licencias de software de soporte? El hardware más antiguo puede tener costes de mantenimiento más elevados que los sistemas modernos. El software de soporte también puede tener altos costes anuales de licencia.**
Interoperabilidad	**¿Existen problemas derivados de la interfaz del sistema con otros sistemas? ¿Los compiladores pueden, por ejemplo, utilizarse con versiones actuales del sistema operativo? ¿Se requiere emulación hardware?**

Figura 21.14 Factores utilizados en la evaluación del entorno.

2. *El número de interfaces de usuario.* Éste es un factor importante en sistemas basados en formularios, en los que cada formulario puede considerarse como una interfaz de usuario independiente. Cuantas más interfaces haya, es más probable que haya inconsistencias y redundancias en dichas interfaces.
3. *El volumen de datos utilizados por el sistema.* Cuanto más alto sea el volumen de datos (número de ficheros, tamaño de la base de datos, etc.), más complejo será el sistema.

Aunque a menudo estos datos son útiles, el obtenerlos puede resultar muy caro y, por lo tanto, poco práctico. Además, no existen valores absolutos que se puedan utilizar. La edad y el tamaño del sistema tienen que tenerse en cuenta cuando se realizan juicios de calidad basados en mediciones.

Idealmente, la evaluación objetiva debería utilizarse para informar de las decisiones sobre lo que hay que hacer con un sistema heredado. Sin embargo, en muchos casos, estas decisiones no son realmente objetivas, sino que se basan en consideraciones políticas u organizacionales. Por ejemplo, si dos empresas se fusionan, la que sea más poderosa políticamente por lo general mantendrá sus sistemas y desechará el resto de los sistemas. Si los gestores senior de una organización deciden cambiar a una nueva plataforma hardware, entonces esto puede requerir el reemplazo de aplicaciones. Si no hay presupuesto disponible para la transformación del sistema en un año concreto, entonces el mantenimiento del sistema puede continuar incluso aunque esto dé lugar a costes elevados a largo plazo.

Comprensibilidad	¿Es muy difícil comprender el código fuente del sistema actual? ¿Son muy complejas las estructuras de control que utiliza? ¿Las variables tienen nombres significativos que reflejan su función?
Documentación	¿Qué documentación del sistema está disponible? ¿La documentación es completa, consistente y actual?
Datos	¿Existe un modelo de datos explícito del sistema? ¿Hasta qué punto están los datos duplicados entre los ficheros? ¿Los datos utilizados por el sistema están actualizados y son consistentes?
Rendimiento	¿El rendimiento de la aplicación es el adecuado? ¿Los problemas de rendimiento tienen un efecto significativo en los usuarios del sistema?
Lenguaje de programación	¿Están disponibles compiladores modernos para el lenguaje de programación utilizado para desarrollar el sistema? ¿El lenguaje de programación todavía se utiliza para el desarrollo de nuevos sistemas?
Gestión de configuraciones	¿Todas las versiones de todas las partes del sistema están gestionadas por un sistema de gestión de configuraciones? ¿Existe una descripción explícita de las versiones de los componentes utilizadas en el sistema actual?
Datos de prueba	¿Existen datos de pruebas para el sistema? ¿Existe un registro de las pruebas de regresión llevadas a cabo cuando se han añadido nuevas características al sistema?
Habilidades del personal	¿Las personas disponibles tienen las habilidades para mantener la aplicación? ¿Existe sólo un número limitado de personas que comprenda el sistema?

Figura 21.15 Factores utilizados en la evaluación de la aplicación.

PUNTOS CLAVE

- El desarrollo del software y su evolución deberían ser un único proceso iterativo integrado que pueda representarse utilizando un modelo en espiral.
- Las leyes de Lehman, como la noción de que el cambio es continuo, describen varias observaciones a partir de estudios a largo plazo de la evolución de los sistemas.
- Existen tres tipos de mantenimiento del software: corrección de errores, modificación del software para trabajar en un nuevo entorno, e implementación de requerimientos nuevos o cambios en éstos.
- Para sistemas a medida, los costes de mantenimiento del software generalmente exceden a los costes de desarrollo del software.
- El proceso de la evolución del software está conducido por peticiones de cambio e incluye análisis del impacto de los cambios, planificación de la entrega e implementación de los cambios.
- La reingeniería del software se refiere a la reestructuración y redocumentación del software para hacerlo más mantenible y más fácil de cambiar.
- El valor de negocio de un sistema heredado y la calidad del software de las aplicaciones y su entorno deberían ser evaluados para determinar si el sistema tiene que ser reemplazado, transformado o mantenido.

LECTURAS ADICIONALES

Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Este excelente libro trata cuestiones generales del mantenimiento y evolución del software así como de la migración de sistemas heredados. El libro se basa en un gran caso de estudio de la transformación de un sistema COBOL a un sistema cliente-servidor basado en Java. (R. C. Seacord *et al.*, 2003, Addison-Wesley.)

The Renaissance of Legacy Systems. Este libro se refiere en su mayor parte a los métodos para hacer evolucionar los sistemas heredados. Sin embargo, incluye una buena exposición general de estos sistemas, casos de estudio que ilustran las estructuras de los sistemas heredados y un capítulo sobre la evaluación de sistemas. (I. Warren, 1998, Springer).

EJERCICIOS

21.1 Explique por qué un sistema software que se usa en un entorno real debe cambiar o se volverá progresivamente menos útil.

21.2 Explique la razón de ser subyacente de las leyes de Lehman. ¿En qué circunstancias podrían no ser aplicables estas leyes?

21.3 Describa brevemente los tres tipos de mantenimiento del software. ¿Por qué es difícil a veces distinguirlos entre sí?

21.4 A usted, como gestor de proyectos software en una compañía especializada en el desarrollo de software para la industria petrolera cerca de la costa, se le ha encomendado la tarea de descubrir los factores que afectan a la mantenibilidad de los sistemas desarrollados por su compañía. Sugiera cómo podría establecer un programa para analizar el proceso de mantenimiento y descubrir métricas de mantenibilidad adecuadas para su compañía.

21.5 A partir de la Figura 21.7, usted puede ver que el análisis de impacto es un subproceso importante en el proceso de evolución del software. Mediante un diagrama, sugiera qué actividades deberían implicarse en el análisis del impacto de los cambios.

21.6 ¿Cuáles son los principales factores que afectan a los costes de reingeniería de los sistemas?

21.7 ¿Cuáles son las condiciones esenciales para que la reingeniería del software tenga éxito?

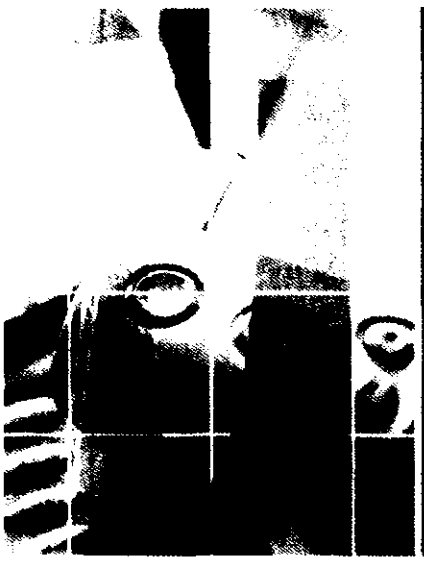
21.8 Indique en qué circunstancias podría una organización decidir desechar un sistema cuando la evaluación de dicho sistema sugiere que tiene una alta calidad y un alto valor de negocio.

21.9 ¿Cuáles son las opciones estratégicas para la evolución de sistemas heredados? ¿Cuándo podría usted normalmente reemplazar todo o parte de un sistema en lugar de continuar manteniendo el software (con o sin reingeniería)?

21.10 Explique por qué los problemas con el software de soporte podrían implicar que una organización tenga que reemplazar sus sistemas heredados.

21.11 ¿Tienen los ingenieros software una responsabilidad profesional para producir código que esté preparado para evolucionar incluso si esto no está explícitamente solicitado por sus clientes?

21.12 La gestión de una organización le ha pedido que lleve a cabo una evaluación del sistema y le sugiere que le gustaría que los resultados de dicha evaluación mostrasen que el sistema es obsoleto y que debería ser reemplazado por un nuevo sistema. Esto significará que varios mantenedores del sistema perderán su trabajo. Su evaluación realmente muestra que el sistema está bien mantenido y que tiene una alta calidad y un alto valor de negocio. ¿Cómo podría informar de estos resultados a la gestión de la organización?



PARTE

5 VERIFICACIÓN Y VALIDACIÓN

Probar un programa es la forma más común de comprobar que satisface su especificación y hace lo que el cliente quiere que haga. Sin embargo, las pruebas sólo son una de las técnicas de verificación y validación. Algunas de estas técnicas, tales como las inspecciones de programas, han sido utilizadas durante casi treinta años, pero todavía no se han convertido en tendencias principales de la ingeniería del software.

En esta parte del libro, se tratan las aproximaciones para verificar que el software satisface su especificación y validar que también satisface las necesidades del cliente del software. Esta parte del libro tiene tres capítulos relacionados con diferentes aspectos de la verificación y validación:

1. El Capítulo 22 presenta una visión general de las aproximaciones para la verificación y validación. Se explica la distinción entre verificación y validación, y el proceso de planificación de la V & V. A continuación, se describen las técnicas estáticas para la verificación de los sistemas. Éstas son técnicas en las que se comprueba el código fuente del programa en lugar de probar su ejecución. Se estudian las inspecciones de programas, el uso de análisis estático automatizado y, finalmente, el rol de los métodos formales en el proceso de verificación.

2. La prueba de programas es el tema del Capítulo 23. Se explica cómo las pruebas se llevan a cabo normalmente en diferentes niveles y se muestran las diferencias entre pruebas de componentes y pruebas del sistema. Utilizando ejemplos sencillos, se introducen varias de las técnicas que pueden utilizarse para diseñar casos de prueba para los programas y, por último, se expone brevemente la automatización de las pruebas. La automatización de las pruebas es el uso de herramientas software que ayudan a reducir el tiempo y el esfuerzo implicado en los procesos de pruebas.

3. El Capítulo 24 trata el tema más especializado de validación de sistemas críticos. Para los sistemas críticos, se tiene que probar a un cliente o a un regulador externo que el sistema satisface su especificación y requerimientos de confiabilidad. Se describen las aproximaciones para la evaluación de la fiabilidad, seguridad y protección, y se explica cómo se puede utilizar la evidencia en los procesos de V & V del sistema para el desarrollo de un caso de prueba de la confiabilidad del sistema.

22 Verificación y validación

Objetivos

El objetivo de este capítulo es introducir la verificación y validación del software con especial énfasis en las técnicas de verificación estática. Cuando haya leído este capítulo:

- comprenderá las diferencias entre verificación y validación del software;
- habrá sido introducido en las inspecciones de programas como un método para descubrir defectos en los programas;
- comprenderá qué es el análisis estático automatizado y cómo se utiliza en verificación y validación;
- comprenderá cómo se utiliza la verificación estática en el proceso de desarrollo de Sala Limpia.

Contenidos

Durante y después del proceso de implementación, el programa que se está desarrollando debe ser comprobado para asegurar que satisface su especificación y entrega la funcionalidad esperada por las personas que pagan por el software. *La verificación y la validación* (V & V) es el nombre dado a estos procesos de análisis y pruebas. La verificación y la validación tienen lugar en cada etapa del proceso del software. V & V comienza con revisiones de los requerimientos y continúa con revisiones del diseño e inspecciones de código hasta la prueba del producto.

La verificación y la validación no son lo mismo, aunque a menudo se confunden. Boehm (Boehm, 1979) expresó de forma sucinta la diferencia entre ellas:

- «Validación: ¿Estamos construyendo el producto correcto?»
- «Verificación: ¿Estamos construyendo el producto correctamente?»

Estas definiciones nos dicen que el papel de la verificación implica comprobar que el software está de acuerdo con su especificación. Debería comprobarse que satisface sus requerimientos funcionales y no funcionales. La validación, sin embargo, es un proceso más general. El objetivo de la validación es asegurar que el sistema software satisface las expectativas del cliente. Va más allá de la comprobación de que el sistema satisface su especificación para demostrar que el software hace lo que el cliente espera que haga. Tal y como se expone en la Parte 2, las especificaciones del sistema software no siempre reflejan los deseos o necesidades reales de los usuarios y los propietarios del sistema.

El objetivo último del proceso de verificación y validación es establecer la seguridad de que el sistema software está «hecho para un propósito». Esto significa que el sistema debe ser lo suficientemente bueno para su uso pretendido. El nivel de confianza requerido depende del propósito del sistema, las expectativas de los usuarios del sistema y el entorno de mercado actual del sistema:

1. *Función del software.* El nivel de confianza requerido depende de lo crítico que sea el software para una organización. Por ejemplo, el nivel de confianza requerido para el software que se utiliza para controlar un sistema de seguridad crítico es mucho más alto que el requerido para un prototipo de un sistema software que ha sido desarrollado para demostrar algunas ideas nuevas.
2. *Expectativas del usuario.* Una reflexión lamentable sobre la industria del software es que muchos usuarios tienen pocas expectativas sobre su software y no se sorprenden cuando éste falla durante su uso. Están dispuestos a aceptar estos fallos del sistema cuando los beneficios de su uso son mayores que sus desventajas. Sin embargo, la tolerancia de los usuarios a los fallos de los sistemas está decreciendo desde los años 90. Actualmente es menos aceptable entregar sistemas no fiables, por lo que las compañías de software deben invertir más esfuerzo para verificar y validar.
3. *Entorno de mercado.* Cuando un sistema se comercializa, los vendedores del sistema deben tener en cuenta los programas competidores, el precio que sus clientes están dispuestos a pagar por el sistema y la agenda requerida para entregar dicho sistema. Cuando una compañía tiene pocos competidores, puede decidir entregar un programa antes de que haya sido completamente probado y depurado, debido a que quiere ser el primero en el mercado. Cuando los clientes no están dispuestos a pagar precios altos por el software, pueden estar dispuestos a tolerar más defectos en él. Todos estos factores pueden considerarse cuando se decide cuánto esfuerzo debería invertirse en el proceso de V & V.

Dentro del proceso de V & V, existen dos aproximaciones complementarias para el análisis y comprobación de los sistemas:

1. *Las inspecciones de software* analizan y comprueban las representaciones del sistema tales como el documento de requerimientos, los diagramas de diseño y el código fuente del programa. Puede usarse las inspecciones en todas las etapas del proceso. Las inspecciones pueden ser complementadas con algún tipo de análisis automático del código fuente de un sistema o de los documentos asociados. Las inspecciones de software y los análisis automáticos son técnicas de V & V estáticas, ya que no se necesita ejecutar el software en una computadora.
2. *Las pruebas del software* implican ejecutar una implementación del software con datos de prueba. Se examinan las salidas del software y su entorno operacional para comprobar que funciona tal y como se requiere. Las pruebas son una técnica dinámica de verificación y validación.

La Figura 22.1 muestra que las inspecciones del software y las pruebas son actividades complementarias en el proceso del software. Las flechas indican las etapas en el proceso en las que pueden utilizarse dichas técnicas. Por lo tanto, se pueden utilizar las inspecciones del software en todas las etapas del proceso de desarrollo. Comenzando por los requerimientos, puede inspeccionarse cualquier representación legible del software. Tal y como se ha indicado, las revisiones de los requerimientos y del diseño son las principales técnicas utilizadas para la detección de errores en el diseño y la especificación.

Sólo puede probarse un sistema cuando está disponible un prototipo o una versión ejecutable del programa. Una ventaja del desarrollo incremental es que una versión probable del sistema está disponible en etapas tempranas del proceso de desarrollo. Las funcionalidades pueden probarse a medida que se van añadiendo al sistema, por lo que no tiene que realizarse una implementación completa antes de que comiencen las pruebas.

Las técnicas de inspección comprenden las inspecciones de programas, el análisis automático del código fuente y la verificación formal. Sin embargo, las técnicas estáticas sólo pueden comprobar la correspondencia entre un programa y su especificación (verificación); no pueden demostrar que el software es operacionalmente útil. Tampoco se pueden utilizar técnicas estáticas para comprobar las propiedades emergentes del software tales como su rendimiento y fiabilidad.

Aunque el uso de las inspecciones del software no es generalizado, la prueba de programas siempre será la principal técnica de verificación y validación. Las pruebas implican ejecutar el programa utilizando datos similares a los datos reales procesados por el programa. Los

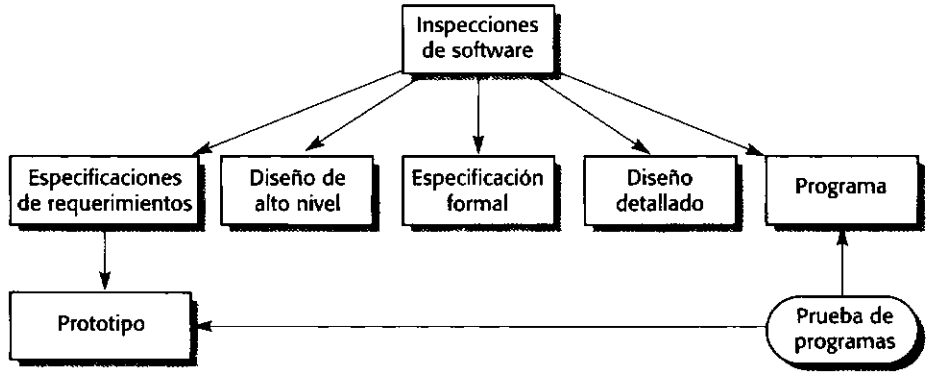


Figura 22.1 Verificación y validación estática y dinámica.

defectos en los programas se descubren examinando las salidas del programa y buscando las anomalías. Existen dos tipos distintos de pruebas que pueden utilizarse en diferentes etapas del proceso del software:

1. *Las pruebas de validación* intentan demostrar que el software es el que el cliente quiere —que satisface sus requerimientos—. Como parte de la prueba de validación, se pueden utilizar pruebas estadísticas para probar el rendimiento y la fiabilidad de los programas, y para comprobar cómo trabaja en ciertas condiciones operacionales. En el Capítulo 24 se analizan las pruebas estadísticas y la estimación de la fiabilidad.
2. *Las pruebas de defectos* intentan revelar defectos en el sistema en lugar de simular su uso operacional. El objetivo de las pruebas de defectos es hallar inconsistencias entre un programa y su especificación. En el Capítulo 23 se tratan las pruebas de defectos.

Por supuesto, no existe un límite perfectamente definido entre estas aproximaciones de pruebas. Durante las pruebas de validación, se encontrarán defectos en el sistema. Durante las pruebas de defectos, alguno de los tests mostrará que el programa satisface sus requerimientos.

Normalmente, los procesos de V & V y depuración se intercalan. A medida que se descubren defectos en el programa que se está probando, tiene que cambiarse éste para corregir tales defectos. Sin embargo, las pruebas (o más generalmente la verificación y validación) y la depuración tienen diferentes objetivos:

1. Los procesos de verificación y validación intentan establecer la existencia de defectos en el sistema software.
2. La depuración es un proceso (Figura 22.2) que localiza y corrige estos defectos.

No existe un método sencillo para la depuración de programas. Los depuradores habilidosos buscan patrones en las salidas de las pruebas en donde se ponen de manifiesto los defectos y utilizan su conocimiento sobre el tipo de defecto, el patrón de salida, el lenguaje de programación y el proceso de programación para localizar el defecto. Durante el proceso de depuración, puede utilizarse el conocimiento de errores comunes de programación (como olvidar incrementar un contador) y hacer corresponder éstos con los patrones observados. También deberían buscarse errores característicos de los lenguajes de programación, tales como errores de direccionamiento de punteros en C.

Localizar los defectos en un programa no siempre es un proceso sencillo, ya que el defecto puede no estar cerca del punto en el que falló el programa. Para localizar un defecto de un programa, se puede tener que diseñar pruebas adicionales que reproduzcan el defecto original y que determinen con precisión su localización en el programa. Se puede tener que hacer manualmente una traza del programa, línea por línea. Las herramientas de depuración que recopilan información sobre la ejecución del programa también pueden ayudar a localizar la fuente de un problema.

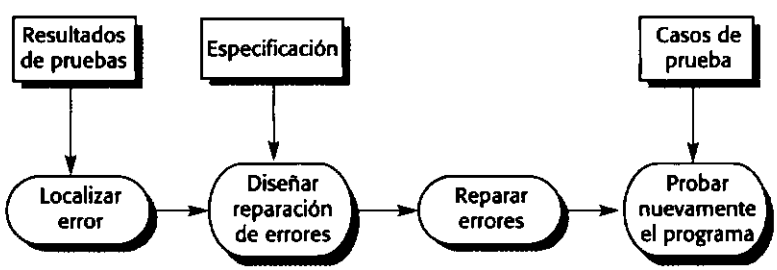


Figura 22.2 El proceso de depuración.

Las herramientas de depuración interactivas generalmente forman parte de un conjunto de herramientas de soporte del lenguaje que se integran con un sistema de compilación. Éstas proporcionan un entorno de ejecución especializado para el programa que permite acceder a la tabla de símbolos del compilador y, desde aquí, a los valores de las variables del programa. Se puede controlar la ejecución «paso a paso» del programa sentencia por sentencia. Después de ejecutar cada sentencia, pueden examinar los valores de las variables y así se puede descubrir la localización del defecto.

Cuando se ha descubierto un defecto en el programa, hay que corregirlo y volver a validar el sistema. Esto puede implicar volver a inspeccionar el programa o hacer pruebas de regresión en las que se ejecutan de nuevo los tests existentes. Las pruebas de regresión se utilizan para comprobar que los cambios en el programa no introducen nuevos defectos. La experiencia ha demostrado que una alta proporción de «reparaciones» de defectos son incompletas o bien introducen nuevos defectos en el programa.

En principio, deberían repetirse todos los tests después de la reparación de cada defecto. En la práctica, esto normalmente supone un coste demasiado elevado. Como parte del plan de pruebas, deberían identificarse dependencias entre los componentes y las pruebas asociadas con cada componente. Esto es, debería poderse establecer una traza entre los casos de prueba y los componentes que son probados. Si esta trazabilidad se documenta, entonces se puede ejecutar un subconjunto de los casos de prueba del sistema para comprobar el componente modificado y sus dependientes.

22.1 Planificación de la verificación y validación

La verificación y validación es un proceso caro. Para algunos sistemas, tales como los sistemas de tiempo real con restricciones no funcionales complejas, más de la mitad del presupuesto para el desarrollo del sistema puede invertirse en V & V. Es necesaria una planificación cuidadosa para obtener el máximo provecho de las inspecciones y pruebas y controlar los costes del proceso de verificación y validación.

Debería comenzarse la planificación de la verificación y validación del sistema en etapas tempranas del proceso de desarrollo. El modelo de proceso de desarrollo del software mostrado en la Figura 22.3 se denomina a veces modelo V (gírese la Figura 22.3 desde su extremo de la derecha para ver la V). Es una instanciación del modelo genérico en cascada (véase

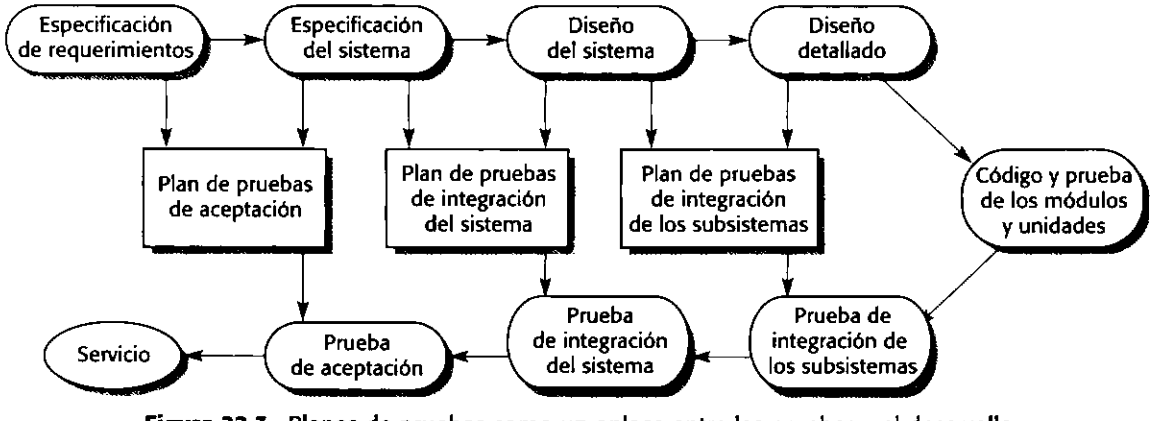


Figura 22.3 Planes de pruebas como un enlace entre las pruebas y el desarrollo.

el Capítulo 4) y muestra que los planes de pruebas deberían derivarse a partir de la especificación y diseño del sistema. Este modelo también divide la V & V del sistema en varias etapas. Cada etapa está conducida por las pruebas que tienen que definirse para comprobar la conformidad del programa con su diseño y especificación.

Como parte del proceso de planificación de V & V, habría que decidir un equilibrio entre las aproximaciones estáticas y dinámicas de la verificación y validación, y pensar en estándares y procedimientos para las inspecciones y pruebas del software, establecer listas de comprobación para conducir las inspecciones de programas (véase la Sección 22.3) y definir el plan de pruebas del software.

El relativo esfuerzo destinado a las inspecciones y las pruebas depende del tipo de sistema a desarrollar y de los expertos de la organización en la inspección de programas. Como regla general, cuanto más crítico sea el sistema, debería dedicarse más esfuerzo a las técnicas de verificación estáticas.

La planificación de las pruebas está relacionada con el establecimiento de estándares para el proceso de las pruebas, no sólo con la descripción de los productos de las pruebas. Los planes de pruebas, además de ayudar a los gestores a asignar recursos y estimar el calendario de las pruebas, son de utilidad para los ingenieros del software implicados en el diseño y la realización de las pruebas del sistema. Éstos ayudan al personal técnico a obtener una panorámica general de las pruebas del sistema y ubicar su propio trabajo en este contexto. Una buena descripción de los planes de pruebas y su relación con los planes de calidad más generales se proporciona en Frewin y Hatton (Frewin y Hatton, 1986). Humphrey (Humphrey, 1989) y Kit (Kit, 1995) también incluyen estudios sobre la planificación de las pruebas.

Los principales componentes de un plan de pruebas para un sistema grande y complejo se muestran en la Figura 22.4. Además de determinar el calendario y procedimientos de las pruebas, el plan de pruebas define los recursos hardware y software que se requieren. Éste es útil para los gestores del sistema que son los responsables de asegurar que estos recur-

El proceso de prueba
Una descripción de las principales fases del proceso de prueba. Éstas podrían describirse como se hizo anteriormente en este capítulo.

Trazabilidad de requerimientos
Los usuarios son los más interesados en que el sistema satisfaga sus requerimientos y las pruebas deberían planificarse para que todos los requerimientos se prueben individualmente.

Elementos probados
Deberían especificarse los elementos del proceso del software que tienen que ser probados.

Calendario de pruebas
Un calendario de todas las pruebas y la asignación de recursos para este calendario se enlaza, obviamente, con la agenda general del desarrollo del proyecto.

Procedimientos de registro de las pruebas
No es suficiente ejecutar simplemente las pruebas; los resultados de las pruebas deben ser registrados sistemáticamente. Debe ser posible auditar el proceso de pruebas para comprobar que se ha llevado a cabo correctamente.

Requerimientos hardware y software
Esta sección debería determinar las herramientas software requeridas y la utilización estimada del hardware.

Restricciones
En esta sección deberían anticiparse las restricciones que afectan al proceso de pruebas como la escasez de personal.

Figura 22.4
La estructura de un plan de pruebas.

sos están disponibles para el equipo de pruebas. Los planes de pruebas normalmente deberían incluir cantidades significativas de contingencias para que los desajustes en la implementación y el diseño puedan solucionarse y el personal pueda ser reasignado a otras actividades.

Para sistemas más pequeños, se puede utilizar un plan de pruebas menos formal, pero sigue siendo necesario un documento formal para soportar la planificación del proceso de pruebas. Para algunos procesos ágiles como la programación extrema, las pruebas son inseparables del desarrollo. Al igual que otras actividades de planificación, la planificación de las pruebas también es incremental. En XP, el cliente es el último responsable de decidir cuánto esfuerzo debería dedicarse a la prueba del sistema.

Los planes de pruebas no son documentos estáticos, sino que evolucionan durante el proceso de desarrollo. Los planes de pruebas cambian debido a retrasos en otras etapas del proceso de desarrollo. Si parte de un sistema está incompleto, el sistema no puede probarse como un todo. Entonces tiene que revisarse el plan de pruebas para volver a desplegar y a asignar a los encargados de las pruebas a alguna otra actividad, y recuperarlos cuando el software vuelva a estar disponible.

22.2 Inspecciones de software

Las inspecciones de software son un proceso de V & V estático en el que un sistema software se revisa para encontrar errores, omisiones y anomalías. Generalmente, las inspecciones se centran en el código fuente, pero puede inspeccionarse cualquier representación legible del software como los requerimientos o un modelo de diseño. Cuando se inspecciona un sistema, se utiliza conocimiento del sistema, su dominio de aplicación y el lenguaje de programación o modelo de diseño para descubrir errores.

Existen tres ventajas fundamentales de la inspección sobre las pruebas:

1. Durante las pruebas, los errores pueden enmascarar (ocultar) otros errores. Cuando se descubre un error, nunca se puede estar seguro de si otras anomalías de salida son debidas a un nuevo error o son efectos laterales del error original. Debido a que la inspección es un proceso estático, no hay que preocuparse de las interacciones entre errores. Por lo tanto, una única sesión de inspección puede descubrir muchos errores en un sistema.
2. Pueden inspeccionarse versiones incompletas de un sistema sin costes adicionales. Si un programa está incompleto, entonces se necesita desarrollar software de soporte especializado para las pruebas a fin de probar aquellas partes que están disponibles. Esto, obviamente, añade costes al desarrollo del sistema.
3. Además de buscar los defectos en el programa, una inspección también puede considerar atributos de calidad más amplios de un programa tales como grado de cumplimiento con los estándares, portabilidad y mantenibilidad. Puede buscarse ineficiencias, algoritmos no adecuados y estilos de programación que podrían hacer que el sistema fuese difícil de mantener y actualizar.

Las inspecciones son una idea antigua. Ha habido varios estudios y experimentos que han demostrado que las inspecciones son más efectivas para descubrir defectos que las pruebas del programa. Fagan (Fagan, 1986) declaró que más del 60% de los errores en un programa pueden detectarse utilizando inspecciones de programa informales. Mills y otros (Mills *et al.*, 1987) sugieren que una aproximación más formal de la inspección basada en la corrección de

los argumentos puede detectar más del 90% de los errores en un programa. Esta técnica se utiliza en el proceso de Sala Limpia descrito en la Sección 22.4. Selby y Basili (Selby *et al.*, 1987) compararon empíricamente la efectividad de las inspecciones y de las pruebas. Observaron que la revisión estática de código era más efectiva y menos costosa que las pruebas de defectos a la hora de encontrar defectos en los programas. Gilb y Graham (Gilb y Graham, 1993) también encontraron que esto era cierto.

Las revisiones y las pruebas tienen cada una sus ventajas e inconvenientes y deberían utilizarse conjuntamente en el proceso de verificación y validación. En realidad, Gilb y Graham sugieren que uno de los usos más efectivos de las revisiones es la revisión de los casos de prueba para un sistema. Las revisiones pueden descubrir problemas con estos tests y ayudar a diseñar formas más efectivas para probar el sistema. Se puede empezar la V & V del sistema con inspecciones en etapas tempranas del proceso de desarrollo, pero una vez que se integra un sistema, se necesita comprobar sus propiedades emergentes y que la funcionalidad del sistema es la que su propietario realmente quiere.

A pesar del éxito de las inspecciones, se ha demostrado que es difícil introducir las inspecciones formales en muchas organizaciones de desarrollo de software. Los ingenieros de software con experiencia en la prueba de programas a menudo son reacios a aceptar que las inspecciones pueden ser más efectivas para detectar defectos que las pruebas. Los gestores pueden ser reacios debido a que las inspecciones requieren costes adicionales durante el diseño y el desarrollo. Pueden no querer asumir el riesgo de que no obtendrán los correspondientes ahorros durante las pruebas de los programas.

No hay duda de que las inspecciones sobrecargan al inicio los costes de V & V del software y conducen a un ahorro de costes sólo después de que los equipos de desarrollo adquieran experiencia en su uso. Además, hay problemas prácticos en cuanto a la organización de las inspecciones: éstas requieren tiempo para organizarse y parecen ralentizar el proceso de desarrollo. Es difícil convencer a un gestor muy presionado que este tiempo puede recuperarse más tarde debido a que se tendrá que emplear menos tiempo depurando el programa.

22.2.1 El proceso de inspección de programas

Las inspecciones de programas son revisiones cuyo objetivo es la detección de defectos en el programa. El concepto de un proceso de inspección formalizado se desarrolló por primera vez por IBM en los años 70 (Fagan, 1976; Fagan, 1986). Actualmente es un método bastante utilizado de verificación de programas, especialmente en ingeniería de sistemas críticos. A partir del método original de Fagan, se han desarrollado varias aproximaciones alternativas de las inspecciones (Gilb y Graham, 1993). Todas ellas están basadas en un grupo con miembros que tienen diferentes conocimientos realizando una revisión cuidadosa línea por línea del código fuente del programa.

La diferencia principal entre las inspecciones de programas y otros tipos de revisiones de calidad es que el objetivo primordial de las inspecciones es encontrar defectos en el programa en lugar de considerar cuestiones de diseño más generales. Los defectos pueden ser errores lógicos, anomalías en el código que podrían indicar una condición errónea, o el incumplimiento de los estándares del proyecto o de la organización. Por otra parte, otros tipos de revisión pueden estar más relacionados con la agenda, los costes, el progreso frente a hitos definidos o la evaluación de si es probable que el software cumpla los objetivos fijados por la organización.

La inspección de programas es un proceso formal realizado por un equipo de al menos cuatro personas. Los miembros del equipo analizan sistemáticamente el código y señalan posibles

Autor o propietario	El programador o diseñador responsable de generar el programa o documento. Responsable de reparar los defectos descubiertos durante el proceso de inspección.
Inspector	Encuentra errores, omisiones e inconsistencias en los programas y documentos. También puede identificar cuestiones más generales que están fuera del ámbito del equipo de inspección.
Lector	Presenta el código o documento en una reunión de inspección.
Secretario	Registra los resultados de la reunión de inspección.
Presidente o moderador	Gestiona el proceso y facilita la inspección. Realiza un informe de los resultados del proceso para el moderador jefe.
Moderador jefe	Responsable de las mejoras del proceso de inspección, actualización de las listas de comprobación, estándares de desarrollo, etc.

Figura 22.5 Roles en el proceso de inspección.

defectos. En las propuestas originales de Fagan, se sugieren roles tales como autor, lector, probador y moderador. El lector lee el código en voz alta al equipo de inspección, el probador inspecciona el código desde una perspectiva de prueba y el moderador organiza el proceso.

A medida que las organizaciones ganan experiencia con la inspección, han surgido otras propuestas para los roles del equipo. En un estudio de cómo la inspección fue introducida con éxito en el proceso de desarrollo de Hewlett-Packard, Grady y Van Slack (Grady y Van Slack, 1994) sugieren seis roles, tal y como se muestra en la Figura 22.5. No creen que sea necesario leer el programa en voz alta. La misma persona puede adoptar más de un rol de forma que el tamaño del equipo puede variar de una inspección a otra. Gilb y Graham sugieren que los inspectores deberían ser seleccionados para reflejar diferentes puntos de vista tales como pruebas, usuario final y gestión de la calidad.

Las actividades en el proceso de inspección se muestran en la Figura 22.6. Antes de que comience una inspección del programa, es esencial que:

1. Se tenga una especificación precisa del código a inspeccionar. Es imposible inspeccionar un componente a un nivel de detalle requerido para detectar defectos sin una especificación completa.
2. Los miembros del equipo de inspección estén familiarizados con los estándares de la organización.
3. Se haya distribuido una versión compilable y actualizada del código a todos los miembros del equipo. No existe ninguna razón para inspeccionar código que esté «casi completo» incluso si un retraso provoca desfases en la agenda.

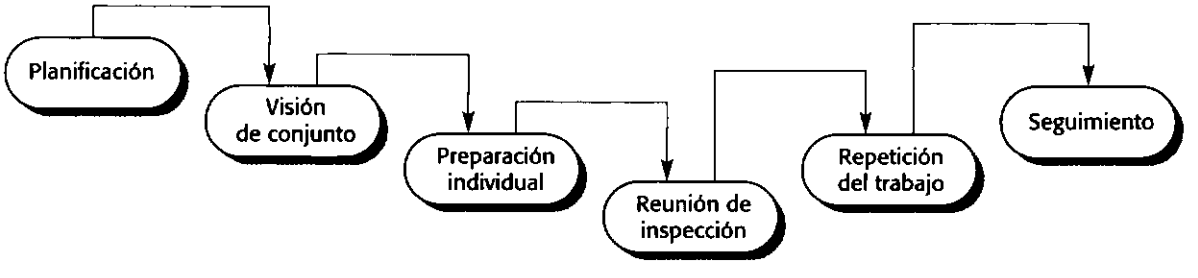


Figura 22.6 El proceso de inspección.

El moderador del equipo de inspección es el responsable de la planificación de la inspección. Esto implica seleccionar un equipo de inspección, organizar una sala de reuniones y asegurar que el material a inspeccionar y sus especificaciones están completas. El programa a inspeccionar se presenta al equipo de inspección durante la etapa de revisión general cuando el autor del código describe lo que el programa debería hacer. A continuación se procede a un periodo de preparación individual. Cada miembro del equipo de inspección estudia la especificación y el programa y busca defectos en el código.

La inspección en sí misma debería ser bastante corta (no más de dos horas) y debería centrarse en la detección de defectos, cumplimiento de los estándares y programación de baja calidad. El equipo de inspección no debería sugerir cómo deben repararse estos defectos ni recomendar cambios en otros componentes.

A continuación de la inspección, el autor del programa debería realizar los cambios para corregir los problemas identificados. En la etapa siguiente, el moderador debería decidir si se requiere una reinspección de código. Puede decidir que no se requiere una reinspección completa y que los defectos han sido reparados con éxito. El programa entonces es aprobado por el moderador para su entrega.

Durante una inspección, a menudo se utiliza una lista de comprobación de errores de programación comunes para centrar el análisis. Esta lista de comprobación puede basarse en ejemplos de listas de comprobación de libros o en conocimiento de los defectos que son comunes en un dominio de aplicación particular. Se necesitan diferentes listas de comprobación para distintos lenguajes de programación debido a que cada lenguaje tiene sus propios errores característicos. Humphrey (Humphrey, 1989), en un extenso estudio sobre las inspecciones, da varios ejemplos de listas de comprobación para inspección.

Esta lista de comprobación varía de acuerdo con el lenguaje de programación debido a los diferentes niveles de comprobación proporcionados por el compilador del lenguaje. Por ejemplo, un compilador Java comprueba que las funciones tienen el número correcto de parámetros; un compilador C no lo hace. En la Figura 22.7 se muestran posibles comprobaciones que podrían realizarse durante un proceso de inspección. Gilb y Graham (Gilb y Graham, 1993) resaltan que cada organización debería desarrollar su propia lista de comprobación de inspecciones basada en estándares y prácticas locales. Las listas de comprobación deberían ser actualizadas regularmente a medida que se encuentran nuevos tipos de defectos.

El tiempo necesario para una inspección y la cantidad de código que puede abarcar dependen de la experiencia del equipo de desarrollo, el lenguaje de programación y el dominio de la aplicación. Tanto Fagan en IBM como Barnard y Price (Barnard y Price, 1994), quienes evaluaron el proceso de inspección para software de telecomunicaciones, llegan a conclusiones similares:

1. Alrededor de 500 sentencias de código fuente por hora pueden presentarse durante la etapa de revisión general.
2. Durante la preparación individual, pueden examinarse alrededor de 125 sentencias de código fuente por hora.
3. Pueden inspeccionarse por hora de 90 a 125 sentencias durante la reunión de inspección.

Con cuatro personas involucradas en un equipo de inspección, el coste de inspeccionar 100 líneas de código es aproximadamente equivalente a un esfuerzo de una persona-día. Esto supone que la inspección en sí misma lleva alrededor de una hora y que cada miembro del equipo emplea de una a dos horas en preparar la inspección. Los costes de las pruebas son muy variables y dependen del número de defectos en el programa. Sin embargo, el esfuerzo re-

Defectos de datos	¿Se inicializan todas las variables antes de que se utilicen sus valores? ¿Tienen nombre todas las constantes? ¿El límite superior de los vectores es igual al tamaño del vector o al tamaño 21? Si se utilizan cadenas de caracteres, ¿tienen un delimitador explícitamente asignado? ¿Existe alguna posibilidad de que el búfer se desborde?
Defectos de control	Para cada sentencia condicional, ¿es correcta la condición? ¿Se garantiza que termina cada bucle? ¿Están puestas correctamente entre llaves las sentencias compuestas? En las sentencias case, ¿se tienen en cuenta todos los posibles casos? Si se requiere una sentencia break después de cada caso en las sentencias case, ¿se ha incluido?
Defectos de entrada/salida	¿Se utilizan todas las variables de entrada? ¿Se les asigna un valor a todas las variables de salida? ¿Pueden provocar corrupciones de datos las entradas no esperadas?
Defectos de interfaz	¿Las llamadas a funciones y a métodos tienen el número correcto de parámetros? ¿Concuerdan los tipos de parámetros reales y formales? ¿Están en el orden correcto los parámetros? Si los componentes acceden a memoria compartida, ¿tienen el mismo modelo de estructura de la memoria compartida?
Defectos de gestión de almacenamiento	Si una estructura enlazada se modifica, ¿se reasignan correctamente todos los enlaces? Si se utiliza almacenamiento dinámico, ¿se asigna correctamente el espacio de memoria? ¿Se desasigna explícitamente el espacio de memoria cuando ya no se necesita?
Defectos de manejo de excepciones	¿Se tienen en cuenta todas las condiciones de error posibles?

Figura 22.7 Comprobaciones de inspección.

querido para la inspección de programas es probablemente menos de la mitad del esfuerzo que se requeriría para una prueba de defectos equivalente.

Algunas organizaciones (Gilb y Graham, 1993) han abandonado actualmente la prueba de componentes en favor de las inspecciones. Han comprobado que las inspecciones de programas son tan efectivas a la hora de encontrar errores que los costes de las pruebas de componentes no son justificables. Estas organizaciones observan que las inspecciones de componentes, combinados con las pruebas del sistema, son la estrategia de V & V más rentable. Tal y como se indica más adelante en el capítulo, esta aproximación se utiliza en el proceso de desarrollo de software de Sala Limpia.

La introducción de las inspecciones tiene implicaciones para la gestión de proyectos. Una gestión sensibilizada es importante si las inspecciones tienen que ser aceptadas por los equipos de desarrollo del software. La inspección de programas es un proceso público de detección de errores comparado con el proceso más privado de prueba de componentes. Inevitablemente, los errores cometidos por individuos se muestran a todo el equipo de programación. Los líderes de los equipos de inspección deben estar capacitados para gestionar el proceso cuidadosamente y desarrollar una cultura que proporcione apoyo cuando se detectan errores y que no exista el sentimiento de culpa asociado a dichos errores.

A medida que una organización gana experiencia en el proceso de las inspecciones, puede utilizar los resultados de éstas para ayudar a la mejora del proceso. Las inspecciones son

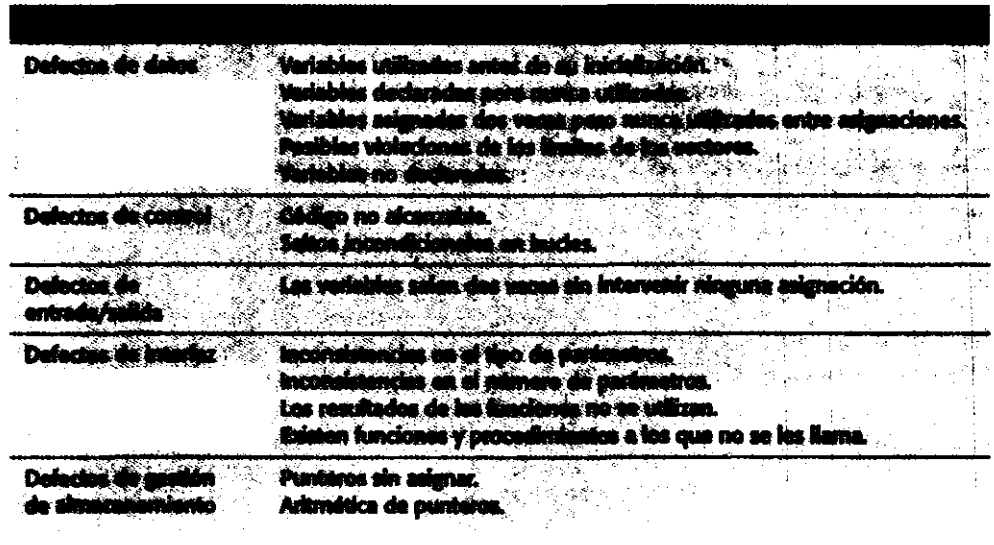
una forma ideal de recopilar datos sobre el tipo de defectos que se producen. El equipo de inspección y los autores del código que fue inspeccionado pueden sugerir razones de por qué se introdujeron estos defectos. En donde sea posible, el proceso debería entonces ser modificado para eliminar las razones de los defectos, de forma que éstos puedan evitarse en sistemas futuros.

22.3 Análisis estático automatizado

Las inspecciones son una forma de análisis estático —se examina el programa sin ejecutarlo—. Tal y como se ha indicado, las inspecciones a menudo están dirigidas por listas de comprobación de errores y heurísticas que identifican errores comunes en diferentes lenguajes de programación. Para algunos errores y heurísticas, es posible automatizar el proceso de comprobación de programas frente a estas listas, las cuales han propiciado el desarrollo de analizadores estáticos automatizados para diferentes lenguajes de programación.

Los analizadores estáticos son herramientas software que escanean el código fuente de un programa y detectan posibles defectos y anomalías. Analizan el código del programa y así reconocen los tipos de sentencias en el programa. Pueden detectar si las sentencias están bien formadas, hacer inferencias sobre el flujo de control del programa y, en muchos casos, calcular el conjunto de todos los posibles valores para los datos del programa. Complementan las facilidades de detección de errores proporcionadas por el compilador del lenguaje. Pueden utilizarse como parte del proceso de inspección o como una actividad separada del proceso V & V.

El objetivo del análisis estático automatizado es llamar la atención del inspector sobre las anomalías del programa, tales como variables que se utilizan sin inicialización, variables que no se usan o datos cuyo valor podría estar fuera de alcance. Algunas de las comprobaciones que se pueden detectar mediante análisis estático se muestran en la Figura 22.8. Las anomalías son a menudo el resultado de errores de programación u omisiones, de forma que resalten aspectos del programa que podrían funcionar mal. Sin embargo, debería comprenderse que estas anomalías no son necesariamente defectos en el programa. Pueden ser deliberadas o pueden no tener consecuencias adversas.



Defectos de datos	Variables utilizadas antes de su inicialización. Variables declaradas pero nunca utilizadas. Variables asignadas dos veces pero nunca utilizadas entre asignaciones. Posibles violaciones de los límites de los vectores. Variables no declaradas.
Defectos de control	Código no alcanzable. Saltos incondicionales en bucles.
Defectos de entrada/salida	Las variables sufren dos veces sin intervenir ninguna asignación.
Defectos de interfaz	Inconsistencias en el tipo de parámetros. Inconsistencias en el número de parámetros. Los resultados de las funciones no se utilizan. Existen funciones y procedimientos a los que no se les llama.
Defectos de gestión de almacenamiento	Punteros sin asignar. Aritmética de punteros.

Figura 22.8 Comprobaciones del análisis estático automatizado.

Las etapas implicadas en el análisis estático comprenden:

1. *Análisis del flujo de control.* Esta etapa identifica y resalta bucles con múltiples salidas o puntos de entrada y código no alcanzable. El código no alcanzable es código que se salta con instrucciones goto no condicionales o que está en una rama de una sentencia condicional en la que la condición nunca es cierta.
2. *Análisis del uso de los datos.* Esta etapa revela cómo se utilizan las variables del programa. Detecta variables que se utilizan sin inicialización previa, variables que se asignan dos veces y no se utilizan entre asignaciones, y variables que se declaran pero nunca se utilizan. El análisis del uso de los datos también descubre pruebas inútiles cuando la condición de prueba es redundante. Las condiciones redundantes son condiciones que son siempre ciertas o siempre falsas.
3. *Análisis de interfaces.* Este análisis comprueba la consistencia de las declaraciones de funciones y procedimientos y su utilización. No es necesario si se utiliza para la implementación un lenguaje fuertemente tipado como Java, ya que el compilador lleva a cabo estas comprobaciones. El análisis de interfaces puede detectar errores de tipos en lenguajes débilmente tipados como FORTRAN y C. El análisis de interfaces también puede detectar funciones y procedimientos que se declaran y nunca son llamados o resultados de funciones que nunca se utilizan.
4. *Análisis del flujo de información.* Esta fase del análisis identifica las dependencias entre las variables de entrada y salida. Mientras no detecte anomalías, muestra cómo se deriva el valor de cada variable del programa a partir de otros valores de variables. Con esta información, una inspección de código debería ser capaz de encontrar valores que han sido calculados erróneamente. El análisis de flujo de información puede también mostrar las condiciones que afectan al valor de una variable.
5. *Análisis de caminos.* Esta fase del análisis semántico identifica todos los posibles caminos en el programa y muestra las sentencias ejecutadas en dicho camino. Esencialmente desenreda el control del programa y permite que cada posible predicado sea analizado individualmente.

Los analizadores estáticos son particularmente valiosos cuando se utiliza un lenguaje de programación como C. Este lenguaje no tiene reglas de tipos estrictas, y la comprobación que puede hacer el compilador de C es limitada. Por lo tanto, es fácil para los programadores cometer errores, y la herramienta de análisis estático puede automáticamente descubrir algunos de los defectos de los programas. Esto es particularmente importante cuando C (y en menor medida C++) se utiliza para desarrollo de sistemas críticos. En este caso, el análisis estático puede descubrir un gran número de errores potenciales y reducir los costes de prueba de forma significativa.

No hay duda de que, para lenguajes como C, el análisis estático es una técnica efectiva para descubrir errores en los programas. Éste compensa los puntos débiles del diseño del lenguaje de programación. Sin embargo, los diseñadores de lenguajes de programación modernos como Java han eliminado algunas características propensas a error. Todas las variables deben ser inicializadas; no hay sentencias goto, de modo que es menos probable crear código inalcanzable de forma accidental, y la gestión del almacenamiento es automática. Esta aproximación para evitar errores en lugar de detectar errores es más efectiva a la hora de mejorar la fiabilidad del programa. Aunque hay disponibles analizadores estáticos para Java, no son ampliamente usados. No está claro si el número de errores detectados justifica el tiempo requerido para analizar su salida.

Por lo tanto, para ilustrar el análisis estático se utiliza un pequeño programa en C en lugar de un programa Java. Los sistemas Unix y Linux incluyen un analizador estático llamado LINT para programas en C. LINT proporciona comprobación estática, que es equivalente a la proporcionada por el compilador en un lenguaje fuertemente tipado como Java. Un ejemplo de la salida producida por LINT se muestra en la Figura 22.9. En esta transcripción de una sesión terminal de UNIX, los comandos se muestran en cursiva. La primera línea de comandos (línea 138) lista el programa. Éste define una función con un parámetro, denominado **printarray**, y a continuación llama a esta función con tres parámetros. Las variables **i** y **c** se declaran, pero nunca se les asigna ningún valor. El valor devuelto por la función nunca se utiliza.

La línea 139 muestra la compilación en C de este programa sin errores obtenido por el compilador de C. A continuación se hace una llamada al analizador estático LINT, que detecta y muestra los errores del programa.

El analizador estático muestra que las variables **c** e **i** han sido utilizadas pero no inicializadas, y que **printarray** ha sido llamado con un número diferente de argumentos que los declarados. También identifica el uso inconsistente del primer argumento en **printarray** y el hecho de que el valor de la función nunca se utiliza.

El análisis basado en herramientas no puede sustituir a las inspecciones, ya que hay algunos tipos de error que los analizadores estáticos no pueden detectar. Por ejemplo, pueden detectar variables no inicializadas, pero no pueden detectar inicializaciones incorrectas. En lenguajes débilmente tipados como C, los analizadores estáticos pueden detectar funciones que tienen números y tipos de argumentos erróneos, pero no pueden detectar situaciones en las que un argumento incorrecto del tipo correcto se ha pasado a una función.

Para tratar algunos de estos problemas, analizadores estáticos tales como LCLint (Orcero, 2000; Evans y Larochelle, 2002) soportan el uso de anotaciones en las que los usuarios definen restricciones y comentarios con estilos en el programa. Estas restricciones permiten a un

```
138% more lint_ex.c

#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Figura 22.9
Análisis estático LINT.

programador especificar qué variables en una función no deberían cambiar, las variables globales a utilizar, y así sucesivamente. El analizador estático puede a continuación comprobar el programa frente a esas restricciones y resaltar secciones de código que parezcan estar incorrectas.

22.4 Verificación y métodos formales

Los métodos formales de desarrollo del software se basan en representaciones matemáticas del software, normalmente como una especificación formal. Estos métodos formales se ocupan principalmente del análisis matemático de la especificación; de transformar la especificación a una representación más detallada semánticamente equivalente; o de verificar formalmente que una representación del sistema es semánticamente equivalente a otra representación.

Usted puede pensar en el uso de métodos formales como la técnica última de verificación estática. Los métodos formales requieren un análisis muy detallado de la especificación del sistema y del programa, y su uso consume a menudo tiempo y resulta caro. Como consecuencia, el uso de métodos formales está restringido principalmente a los procesos de desarrollo de software de seguridad crítico y seguro. El uso de especificaciones matemáticas formales y la verificación asociada fue encargado en los estándares de defensa en el Reino Unido para software de seguridad crítico (MOD, 1995).

Los métodos formales pueden utilizarse en diferentes etapas en el proceso V & V:

1. Puede desarrollarse una especificación formal del sistema y analizarse matemáticamente para buscar inconsistencias. Esta técnica es efectiva para descubrir errores y omisiones de especificación, tal y como se explicó en el Capítulo 10.
2. Puede verificarse formalmente, utilizando argumentos matemáticos, que el código de un sistema software es consistente con su especificación. Esto requiere una especificación formal y es efectiva para descubrir algunos errores de diseño y programación. Puede utilizarse un proceso de desarrollo transformacional o proceso de Sala Limpia en el que una especificación formal se transforma a través de una serie de representaciones más detalladas para soportar el proceso de verificación formal.

El argumento para el uso de la especificación formal y de la verificación del programa asociado es que la especificación formal fuerza un análisis detallado de la especificación. Puede revelar inconsistencias u omisiones potenciales que podrían de otra forma no ser descubiertas hasta que el sistema sea operacional. La verificación formal demuestra que el programa desarrollado satisface su especificación, por lo que los errores de implementación no comprometen la confiabilidad.

El argumento en contra del uso de la especificación formal es que requiere notaciones especializadas. Éstas sólo se pueden utilizar por personal entrenado especialmente y no pueden ser comprendidas por expertos del dominio. Por lo tanto, los problemas con los requerimientos del sistema pueden estar encubiertos por la formalidad. Los ingenieros software no pueden reconocer dificultades potenciales con los requerimientos debido a que no comprenden el dominio; los expertos en el dominio no pueden encontrar estos problemas porque no comprenden la especificación. Aunque la especificación puede ser matemáticamente consistente, puede no especificar las propiedades del sistema que son realmente necesarias.

Verificar un software no trivial consume una gran cantidad de tiempo y requiere herramientas especializadas tales como demostradores de teoremas y expertos matemáticos. Por lo

tanto, es un proceso extremadamente caro y, a medida que el tamaño del sistema crece, los costes de la verificación formal crecen desproporcionadamente. En consecuencia, mucha gente piensa que la verificación formal no es muy rentable. El mismo nivel de confianza en el sistema puede lograrse de forma más económica utilizando otras técnicas de validación como las inspecciones y pruebas de sistemas.

Se ha dicho algunas veces que el uso de métodos formales para el desarrollo de sistemas conduce a sistemas más fiables y seguros. No hay duda de que una especificación formal de un sistema es menos probable que contenga anomalías que tengan que resolverse por el diseñador del sistema. Sin embargo, la especificación formal y la demostración no garantiza que el software será fiable en el uso práctico. Las razones de esto son las siguientes:

1. *La especificación puede no reflejar los requerimientos reales de los usuarios del sistema.* Lutz (Lutz, 1993) descubrió que muchos fallos experimentados por los usuarios eran consecuencia de errores y omisiones en la especificación, que podrían no haberse detectado por una especificación formal del sistema. Además, los usuarios del sistema raramente comprenden las notaciones formales, por lo que no pueden leer directamente la especificación formal para encontrar errores y omisiones.
2. *La demostración puede contener errores.* Las demostraciones de los programas son largas y complejas; por lo tanto, al igual que los programas complejos y grandes, normalmente contienen errores.
3. *La demostración puede asumir un patrón de uso que es incorrecto.* Si el sistema no se usa tal y como se ha anticipado, la demostración puede ser inválida.

A pesar de sus desventajas, la opinión que aquí se formula (expuesta en el Capítulo 10) es que los métodos formales juegan un papel importante en el desarrollo de sistemas software críticos. Las especificaciones formales son muy efectivas descubriendo problemas de la especificación que son las causas más comunes de los fallos de ejecución del sistema. La verificación formal incrementa la confianza en los componentes más críticos de estos sistemas. El uso de aproximaciones formales va en aumento a medida que los clientes lo solicitan y a medida que cada vez más ingenieros están más familiarizados con estas técnicas.

22.4.1 Desarrollo de software de Sala Limpia

Los métodos formales se han integrado con varios procesos de desarrollo del software. En el método B (Wordsworth, 1996), una especificación formal se transforma en un programa a través de una serie de transformaciones que preservan la corrección. SDL (Mitschele-Thiel, 2001) se usa para el desarrollo de sistemas de telecomunicaciones y VDM (Jones, 1986) y Z (Spivey, 1992) han sido utilizados en procesos del tipo en cascada. Otra aproximación bien documentada que utiliza métodos formales es el proceso de desarrollo de Sala Limpia. El desarrollo de software de Sala Limpia (Mills *et al.*, 1987; Cobb y Mills, 1990; Linger, 1994; Prowell *et al.*, 1999) es una filosofía de desarrollo de software que utiliza métodos formales para soportar inspecciones del software rigurosas.

Un modelo del proceso de Sala Limpia se muestra en la Figura 22.10. El objetivo de esta aproximación de desarrollo del software es obtener software con cero defectos. El nombre «Sala Limpia» se derivó de la analogía con la fabricación de unidades de semiconductores en donde los defectos se evitaban mediante su fabricación en una atmósfera ultralimpia. El desarrollo de Sala Limpia es particularmente pertinente en este capítulo, debido a que reemplaza las pruebas de unidades de los componentes del sistema por inspecciones para comprobar la consistencia de estos componentes con sus especificaciones.

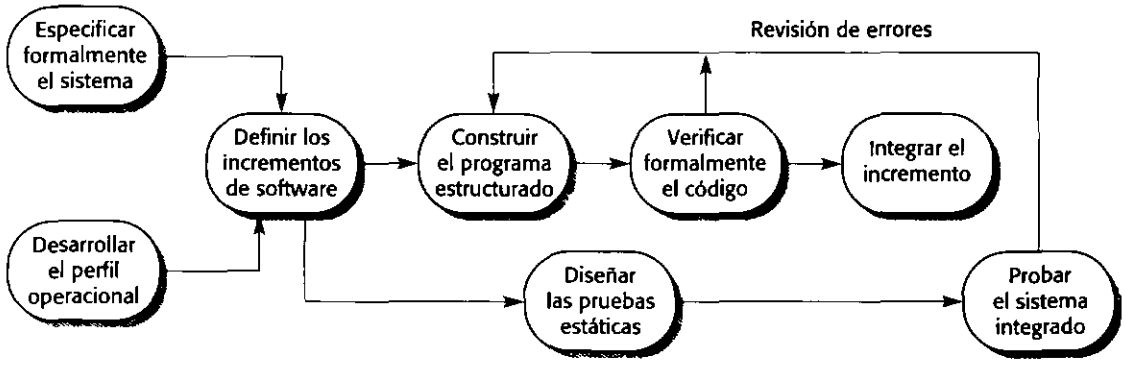


Figura 22.10 El proceso de desarrollo de Sala Limpia.

La aproximación de Sala Limpia al desarrollo del software se basa en cinco estrategias clave:

1. *Especificación formal.* El software a desarrollar se especifica formalmente. Para expresar la especificación se utiliza un modelo de transición de estados que muestra las respuestas del sistema a los estímulos.
2. *Desarrollo incremental.* El software se divide en incrementos que se desarrollan y validan de forma independiente utilizando el proceso de Sala Limpia. Estos incrementos se especifican, con la información de los clientes, en una etapa temprana del proceso.
3. *Programación estructurada.* Se utiliza sólo un número limitado de estructuras de control y de abstracciones de datos. El proceso de desarrollo del programa es un proceso de pasos de refinamiento de la especificación. Se utiliza un número limitado de construcciones y el objetivo es transformar sistemáticamente la especificación para crear el código del programa.
4. *Verificación estática.* El software desarrollado se verifica estáticamente utilizando inspecciones de software rigurosas. No existe ningún proceso de prueba de unidades o módulos para los componentes del código.
5. *Pruebas estadísticas del sistema.* El incremento del software integrado es probado estadísticamente, tal y como se explica en el Capítulo 24, para determinar su fiabilidad. Estas pruebas estadísticas se basan en un perfil operacional, el cual se desarrolla en paralelo con la especificación del sistema, tal y como se muestra en la Figura 22.10.

Existen tres equipos implicados cuando se utiliza el proceso de Sala Limpia para el desarrollo de grandes sistemas:

1. *El equipo de especificación.* Este grupo es responsable del desarrollo y mantenimiento de la especificación del sistema. Este equipo produce especificaciones orientadas al cliente (la definición de requerimientos del usuario) y especificaciones matemáticas para verificación. En algunos casos, cuando la especificación es completa, el equipo de especificación también toma la responsabilidad del desarrollo.
2. *El equipo de desarrollo.* Este equipo tiene la responsabilidad de desarrollar y verificar el software. El software no se ejecuta durante el proceso de desarrollo. Se utiliza una aproximación formal y estructurada para la verificación basada en inspección de código y complementada con argumentos de corrección.
3. *El equipo de certificación.* Este equipo se encarga de desarrollar un conjunto de pruebas estadísticas para ejercitar el software después de que haya sido desarrollado. Es-

tas pruebas se basan en especificación formal. El desarrollo de los casos de prueba se lleva a cabo en paralelo con el desarrollo del software. Los casos de prueba se utilizan para certificar la fiabilidad del software. Los modelos de crecimiento de fiabilidad (Capítulo 24) pueden utilizarse para decidir cuándo parar las pruebas.

El uso de la aproximación de Sala Limpia conduce generalmente a software con muy pocos errores. Coob y Mills analizan varios proyectos de desarrollo con éxito de Sala Limpia que tuvieron una tasa de fallos de funcionamiento muy baja en los sistemas entregados (Coob y Mills, 1990). Los costes de estos proyectos fueron comparables a otros proyectos que usaron técnicas de desarrollo convencionales.

La aproximación al desarrollo incremental en el proceso de Sala Limpia consiste en entregar funcionalidades críticas del cliente en incrementos tempranos. Las funciones del sistema menos importantes se incluyen en incrementos posteriores. Por lo tanto, el cliente tiene la oportunidad de probar estos incrementos críticos antes de que se haya entregado el sistema en su totalidad. Si se descubren problemas con estos incrementos, el cliente comunica esta información al equipo de desarrollo y solicita una nueva entrega del incremento.

Al igual que ocurre con la programación extrema, esto significa que las funciones del cliente más importantes reciben la mayor parte de la validación. A medida que se desarrollan nuevos incrementos, éstos se combinan con los requerimientos existentes y se prueba el sistema integrado. Por lo tanto, los requerimientos existentes se vuelven a probar con nuevos casos de prueba a medida que se añaden nuevos incrementos al sistema.

La inspección rigurosa de programas es una parte fundamental del proceso de Sala Limpia. Se produce un modelo de estados del sistema como una especificación del sistema. Éste se refina a través de una serie de modelos del sistema más detallados hasta conseguir un programa ejecutable. La aproximación utilizada para el desarrollo se basa en transformaciones bien definidas que intentan preservar la corrección de cada transformación a una representación más detallada. En cada etapa se inspecciona la nueva representación, y se desarrollan argumentos matemáticamente rigurosos para demostrar que la salida de la información es consistente con su entrada.

Los argumentos matemáticos utilizados en el proceso de Sala Limpia no son, sin embargo, demostraciones formales de corrección. Las demostraciones matemáticas formales de que un programa es correcto con respecto a su especificación son demasiado caras de llevar a cabo. Dependen del uso del conocimiento sobre la semántica formal del lenguaje de programación para construir teorías que relacionan el programa y su especificación formal. A continuación estas teorías deben probarse matemáticamente, a menudo con la asistencia de grandes y complejos programas de demostradores de teoremas. Debido a su alto coste y a que se necesitan habilidades especiales, las demostraciones se desarrollan normalmente sólo para la mayoría de las aplicaciones de seguridad o de protección críticas.

Se ha observado que la inspección y el análisis formal son muy efectivos en el proceso de Sala Limpia. La inmensa mayoría de los defectos se descubren antes de la ejecución y no se introducen en el software desarrollado. Linger (Linger, 1994) muestra que, en promedio, sólo 2,3 defectos por mil líneas de código fuente fueron descubiertos durante las pruebas para proyectos de Sala Limpia. Los costes totales de desarrollo no se incrementaron debido a que se requirió menos esfuerzo para probar y reparar el software desarrollado.

Selby y otros (Selby *et al.*, 1987), utilizando estudiantes como desarrolladores, llevaron a cabo un experimento que comparaba el desarrollo de Sala Limpia con técnicas convencionales. Comprobaron que la mayoría de los grupos pudieron usar con éxito el método de Sala Limpia. Los programas producidos fueron de mayor calidad que los desarrollados utilizando

técnicas tradicionales; el código fuente tuvo más comentarios y una estructura más simple. Muchos de los equipos de Sala Limpia cumplieron la agenda de desarrollo.

El desarrollo de Sala Limpia funciona bien cuando se practica por ingenieros comprometidos y habilidosos. Los informes sobre el éxito de la aproximación de Sala Limpia en la industria provienen en su mayoría, aunque no de forma exclusiva, de gente que ya lo había utilizado. No se sabe si este proceso puede transferirse de forma efectiva a otros tipos de organizaciones de desarrollo de software. Estas organizaciones pueden tener ingenieros menos comprometidos y menos habilidosos. La transferencia de la aproximación de Sala Limpia, o en realidad de cualquier otra aproximación en la que se utilicen métodos formales, a organizaciones menos avanzadas técnicamente, todavía continúa siendo un reto.

PUNTOS CLAVE

- **La verificación y la validación no son lo mismo. La verificación intenta mostrar que un programa satisface su especificación. La validación intenta mostrar que el programa hace lo que el usuario requiere.**
- **Los planes de pruebas deberían incluir una descripción de los elementos que hay que probar, la agenda de pruebas, los procedimientos para gestionar el proceso de pruebas, los requerimientos hardware y software, y cualquier problema de pruebas que probablemente pueda surgir.**
- **Las técnicas de verificación estática implican examinar y analizar el código fuente del programa para detectar errores. Deberían utilizarse con las pruebas de programas como parte del proceso V & V.**
- **Las inspecciones de programas son efectivas para encontrar errores en los programas. El objetivo de una inspección es localizar defectos. Una lista de comprobación de defectos debería conducir el proceso de la inspección.**
- **En una inspección de programas, un grupo pequeño comprueba el código de forma sistemática. Los miembros del equipo incluyen un líder del equipo o moderador, el autor del código, un lector que presenta el código durante la inspección y un probador que considera el código desde una perspectiva de pruebas.**
- **Los analizadores estáticos son herramientas software que procesan un código fuente de un programa y ponen de manifiesto anomalías tales como secciones de código no utilizadas y variables sin inicializar. Estas anomalías pueden ser el resultado de defectos en el código.**
- **El desarrollo de software de Sala Limpia se centra en técnicas estáticas para la verificación de programas y pruebas estadísticas para la certificación de la fiabilidad del sistema. Se ha utilizado con éxito en la producción de sistemas que tienen un alto nivel de fiabilidad.**

LECTURAS ADICIONALES

Software Quality Assurance: From Theory to Implementation. Este libro proporciona una buena lectura de base sobre la verificación y validación, con un capítulo particularmente bueno sobre revisiones e inspecciones. (D. Galin, 2004, Addison-Wesley.)

«Software inspection». Un número especial de una revista que contiene varios artículos sobre inspección de programas, incluyendo una discusión del uso de dicha técnica con desarrollo orientado a objetos. [*IEEE Software*, 20(4), julio-agosto de 2003.]

«Software debugging, testing and verification». Éste es un artículo general sobre verificación y validación y uno de los pocos artículos que tratan las técnicas de pruebas y verificación estática. [B. Halipern y P. Santhanam, *IBM Systems Journal*, 41(1), enero de 2002.]

Cleanroom Software Engineering: Technology and Process. Un buen libro sobre la aproximación de Sala Limpia que contiene secciones sobre los fundamentos de dicha técnica, el proceso y un caso de estudio práctico. (S. J. Powell *et al.*, 1999, Addison-Wesley.)

EJERCICIOS

22.1 Señale las diferencias entre verificación y validación, y explique por qué la validación es un proceso particularmente difícil.

22.2 Explique por qué no es necesario que un programa esté completamente libre de defectos antes de que sea entregado a sus clientes. ¿Hasta dónde se pueden utilizar las pruebas para validar que el programa cumple con su propósito?

22.3 El plan de pruebas de la Figura 22.4 ha sido diseñado para sistemas a medida que tienen un documento de requerimientos independiente. Sugiera cómo podría modificarse la estructura del plan de pruebas para probar productos software comerciales.

22.4 Explique por qué las inspecciones de programas son una técnica efectiva para descubrir errores en un programa. ¿Qué tipos de errores probablemente no sean descubiertos a través de las inspecciones?

22.5 Sugiera por qué una organización con una cultura elitista y competitiva podría probablemente encontrar difícil introducir las inspecciones de programas como una técnica de V & V.

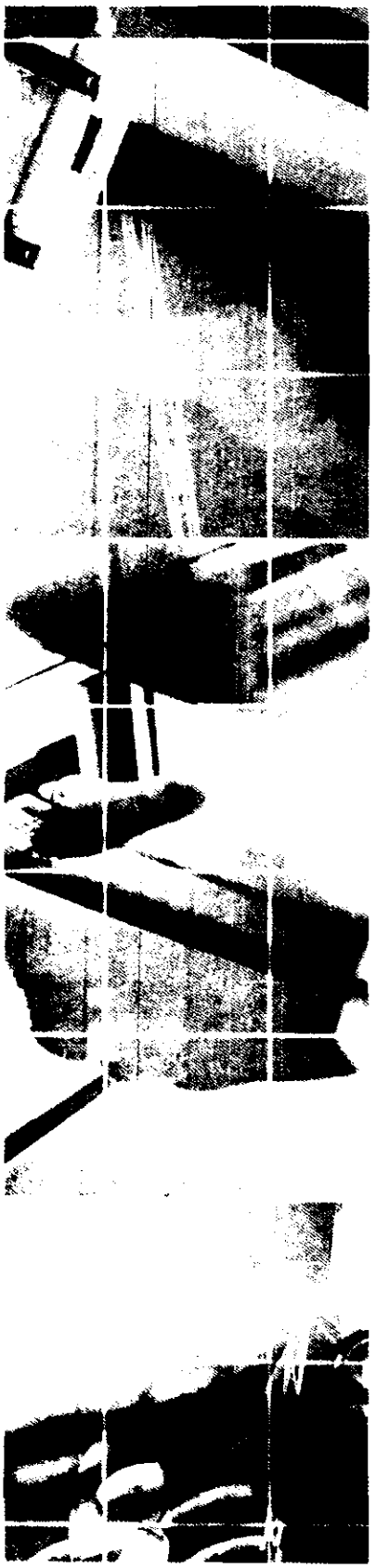
22.6 Utilizando su conocimiento de Java, C++, C o cualquier otro lenguaje de programación, derive una lista de comprobación de errores comunes (no errores sintácticos) que podrían no ser detectados por un compilador, pero que podrían ser detectados en una inspección de programas.

22.7 Genere una lista de condiciones que podrían ser detectadas por un analizador estático para Java, C++ u otro lenguaje de programación que usted utilice. Comente esta lista comparada con la lista dada en la Figura 22.7.

22.8 Explique por qué puede ser rentable utilizar métodos formales en el desarrollo de sistemas software de seguridad críticos. ¿Por qué piensa usted que algunos desarrolladores de este tipo de sistemas están en contra del uso de los métodos formales?

22.9 Un gestor decide utilizar los informes de las inspecciones de programas como entrada para el proceso de valoración del personal. Estos informes muestran quién hace y quién descubre los errores en los programas. ¿Es éste un comportamiento de gestión ético? ¿Podría ser ético si el personal fuese informado con antelación de que esto podría ocurrir? ¿Qué diferencia se podría generar en el proceso de inspección?

22.10 Una aproximación comúnmente adoptada para las pruebas del sistema es probar el sistema hasta que se agote el presupuesto de pruebas y entonces se entrega el sistema a los clientes. Comente la ética de esta aproximación.



23 Pruebas del software

Objetivos

El objetivo de este capítulo es describir el proceso de las pruebas del software e introducir varias técnicas de pruebas. Cuando haya leído este capítulo:

- comprenderá las diferencias entre pruebas de validación y pruebas de defectos;
- comprenderá los principios de las pruebas del sistema y las pruebas de componentes;
- comprenderá tres estrategias que pueden utilizarse para generar casos de pruebas del sistema;
- comprenderá las características esenciales de las herramientas software que soportan la automatización de las pruebas.

Contenidos

En el Capítulo 4 se describió un proceso general de pruebas que comenzaba con la prueba de unidades de programas individuales tales como funciones u objetos. A continuación, éstas se integraban en subsistemas y sistemas, y se probaban las interacciones entre estas unidades. Finalmente, después de entregar el sistema, el cliente puede llevar a cabo una serie de pruebas de aceptación para comprobar que el sistema funciona tal y como se ha especificado.

Este modelo de proceso de pruebas es apropiado para el desarrollo de sistemas grandes; pero para sistemas más pequeños, o para sistemas que se desarrollan mediante el uso de *scripts* o reutilización, a menudo se distinguen menos etapas en el proceso. Una visión más abstracta de las pruebas del software se muestra en la Figura 23.1. Las dos actividades fundamentales de pruebas son la prueba de componentes —probar las partes del sistema— y la prueba del sistema —probar el sistema como un todo.

El objetivo de la etapa de la prueba de componentes es descubrir defectos probando componentes de programas individuales. Estos componentes pueden ser funciones, objetos o componentes reutilizables, tales como los descritos en el Capítulo 19. Durante las pruebas del sistema, estos componentes se integran para formar subsistemas o el sistema completo. En esta etapa, la prueba del sistema debería centrarse en establecer que el sistema satisface sus requerimientos funcionales y no funcionales, y no se comporta de forma inesperada. Inevitablemente, los defectos en los componentes que no se han detectado durante las primeras etapas de las pruebas se descubren durante las pruebas del sistema.

Tal y como se ha explicado en el Capítulo 22, el proceso de pruebas del software tiene dos objetivos distintos:

1. *Para demostrar al desarrollador y al cliente que el software satisface sus requerimientos.* Para el software a medida, esto significa que debería haber al menos una prueba para cada requerimiento de los documentos de requerimientos del sistema y del usuario. Para productos de software genéricos, significa que debería haber pruebas para todas las características del sistema que se incorporarán en la entrega del producto. Tal y como se explicó en el Capítulo 4, algunos sistemas pueden tener una fase de pruebas de aceptación explícita en la que el cliente comprueba formalmente que el sistema entregado cumple su especificación.
2. *Para descubrir defectos en el software en que el comportamiento de éste es incorrecto, no deseable o no cumple su especificación.* La prueba de defectos está relacionada con la eliminación de todos los tipos de comportamientos del sistema no deseables, tales como caídas del sistema, interacciones no permitidas con otros sistemas, cálculos incorrectos y corrupción de datos.

El primer objetivo conduce a las pruebas de validación, en las que se espera que el sistema funcione correctamente usando un conjunto determinado de casos de prueba que reflejan el uso esperado de aquél. El segundo objetivo conduce a la prueba de defectos, en los que los casos de prueba se diseñan para exponer los defectos. Los casos de prueba pueden ser deliberadamente oscuros y no necesitan reflejar cómo se utiliza normalmente el sistema. Para las pruebas de validación, una prueba con éxito es aquella en la que el sistema funciona correc-

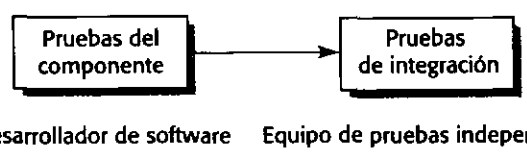


Figura 23.1
Fases de pruebas.

tamente. Para las pruebas de defectos, una prueba con éxito es aquella que muestra un defecto que hace que el sistema funciona incorrectamente.

Las pruebas no pueden demostrar que el software está libre de defectos o que se comportará en todo momento como está especificado. Siempre es posible que una prueba que se haya pasado por alto pueda descubrir problemas adicionales con el sistema. Como dijo de forma elocuente Edsger Dijkstra, una de las primeras figuras líderes en el desarrollo de la ingeniería del software (Dijkstra *et al.*, 1972), «las pruebas sólo pueden demostrar la presencia de errores, no su ausencia».

Generalmente, por lo tanto, el objetivo de las pruebas del software es convencer a los desarrolladores del sistema y a los clientes de que el software es lo suficientemente bueno para su uso operacional. La prueba es un proceso que intenta proporcionar confianza en el software.

Un modelo general del proceso de pruebas se muestra en la Figura 23.2. Los casos de prueba son especificaciones de las entradas para la prueba y la salida esperada del sistema mas una afirmación de lo que se está probando. Los datos de prueba son las entradas que han sido ideadas para probar el sistema. Los datos de prueba a veces pueden generarse automáticamente. La generación automática de casos de prueba es imposible. Las salidas de las pruebas sólo pueden predecirse por personas que comprenden lo que debería hacer el sistema.

Las pruebas exhaustivas, en las que cada posible secuencia de ejecución del programa es probada, son imposibles. Las pruebas, por lo tanto, tienen que basarse en un subconjunto de posibles casos de prueba. Idealmente, algunas compañías deberían tener políticas para elegir este subconjunto en lugar de dejar esto al equipo de desarrollo. Estas políticas podrían basarse en políticas generales de pruebas, tal como una política en la que todas las sentencias de los programas deberían ejecutarse al menos una vez. De forma alternativa, las políticas de pruebas pueden basarse en la experiencia de uso del sistema y pueden centrarse en probar las características del sistema operacional. Por ejemplo:

1. Deberían probarse todas las funciones del sistema a las que se accede a través de menús.
2. Deben probarse todas las combinaciones de funciones (por ejemplo, formateado de textos) a las que se accede a través del mismo menú.
3. En los puntos del programa en los que el usuario introduce datos, todas las funciones deben probarse con datos correctos e incorrectos.

A partir de la experiencia con los principales productos de software tales como procesadores de texto u hojas de cálculo, está claro que durante el uso del producto se utilizan normalmente guías similares durante las pruebas de los productos. Cuando se usan las características del software por separado, éstas normalmente funcionan. Los problemas surgen, tal y como explica Whittaker (Whittaker, 2002), cuando no se han probado conjuntamente combinaciones de características. Él pone el ejemplo de cómo, en un procesador de texto común-

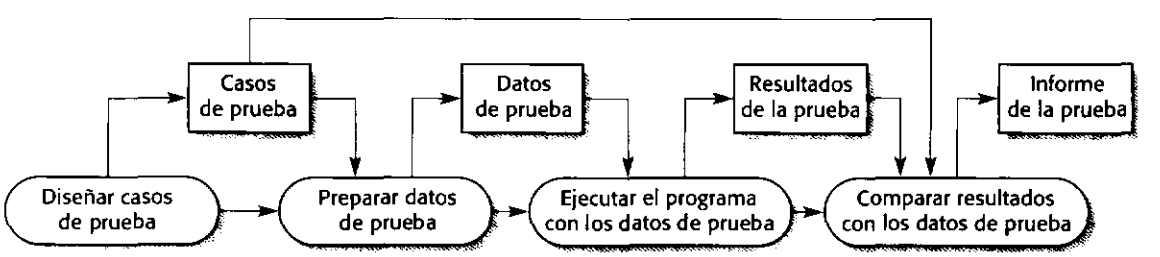


Figura 23.2 Un modelo del proceso de pruebas del software.

mente usado, el uso de notas al pie de página con un formato a dos columnas provoca un formateado incorrecto del texto.

Como una parte del proceso V & V, los gestores tienen que tomar decisiones sobre quién debería ser responsable de las diferentes etapas de las pruebas. Para la mayoría de los sistemas, los programadores tienen la responsabilidad de probar los componentes que ellos han desarrollado. Una vez que lo hacen, el trabajo se pasa a un equipo de integración que integra los módulos de diferentes desarrolladores, construye el software y prueba el sistema como un todo. Para sistemas críticos, puede utilizarse un proceso más formal en el que probadores independientes son responsables de todas las etapas del proceso de prueba. En pruebas de sistemas críticos, las pruebas se desarrollan de forma independiente y se mantienen informes detallados de los resultados de las mismas.

Las pruebas de componentes realizadas por los desarrolladores se basan normalmente en una comprensión intuitiva de cómo los componentes deberían operar. Las pruebas del sistema, sin embargo, tienen que basarse en una especificación escrita del sistema. Ésta puede ser una especificación detallada de requerimientos del sistema, tal y como se indicó en el Capítulo 6, o puede ser una especificación orientada al usuario de más alto nivel de las características que debería implementar el sistema. Normalmente un grupo independiente es responsable de las pruebas del sistema. Tal y como se explicó en el Capítulo 4, el equipo de pruebas del sistema trabaja a partir de los documentos de requerimientos del sistema y del usuario para desarrollar los planes de pruebas del sistema (véase la Figura 4.10).

La mayoría de los tratamientos de pruebas comienzan con las pruebas de componentes y a continuación se realizan las pruebas del sistema. Se ha invertido de forma deliberada el orden de la exposición en este capítulo debido a que cada vez más el desarrollo del software implica integrar componentes reutilizables y configurar y adaptar software existente para satisfacer requerimientos específicos. Todas las pruebas en tales casos son pruebas del sistema, y no hay un proceso separado de pruebas de componentes.

23.1 Pruebas del sistema

Las pruebas del sistema implican integrar dos o más componentes que implementan funciones del sistema o características y a continuación se prueba este sistema integrado. En un proceso de desarrollo iterativo, las pruebas del sistema se ocupan de probar un incremento que va a ser entregado al cliente; en un proceso en cascada, las pruebas del sistema se ocupan de probar el sistema completo.

Para la mayoría de los sistemas complejos, existen dos fases distintas de pruebas del sistema:

1. *Pruebas de integración*, en las que el equipo de pruebas tiene acceso al código fuente del sistema. Cuando se descubre un problema, el equipo de integración intenta encontrar la fuente del problema e identificar los componentes que tienen que ser depurados. Las pruebas de integración se ocupan principalmente de encontrar defectos en el sistema.
2. *Pruebas de entregas*, en las que se prueba una versión del sistema que podría ser entregada a los usuarios. Aquí, el equipo de pruebas se ocupa de validar que el sistema satisface sus requerimientos y con asegurar que el sistema es confiable. Las pruebas de entregas son normalmente pruebas de «caja negra» en las que el equipo de pruebas

se ocupa simplemente de demostrar si el sistema funciona o no correctamente. Los problemas son comunicados al equipo de desarrollo cuyo trabajo es depurar el programa. Cuando los clientes se implican en las pruebas de entregas, éstas a menudo se denominan *pruebas de aceptación*. Si la entrega es lo suficientemente buena, el cliente puede entonces aceptarla para su uso.

Fundamentalmente, se puede pensar en las pruebas de integración como las pruebas de sistemas incompletos compuestos por grupos de componentes del sistema. Las pruebas de entregas consisten en probar la entrega del sistema que se pretende proporcionar a los clientes. Naturalmente, éstas se solapan, en especial cuando se utiliza desarrollo incremental y el sistema para entregar está incompleto. Generalmente, la prioridad en las pruebas de integración es descubrir defectos en el sistema, y la prioridad en las pruebas del sistema es validar que el sistema satisface sus requerimientos. Sin embargo, en la práctica, hay una parte de prueba de validación y una parte de prueba de defectos durante ambos procesos.

23.1.1 Pruebas de integración

El proceso de la integración del sistema implica construir éste a partir de sus componentes (véase el Capítulo 29) y probar el sistema resultante para encontrar problemas que pueden surgir debido a la integración de los componentes. Los componentes que se integran pueden ser componentes comerciales, componentes reutilizables que han sido adaptados a un sistema particular, o componentes nuevos desarrollados. Para muchos sistemas grandes, es probable que se usen los tres tipos de componentes. Las pruebas de integración comprueban que estos componentes realmente funcionan juntos, son llamados correctamente y transfieren los datos correctos en el tiempo preciso a través de sus interfaces.

La integración del sistema implica identificar grupos de componentes que proporcionan alguna funcionalidad del sistema e integrar éstos añadiendo código para hacer que funcionen conjuntamente. Algunas veces, primero se desarrolla el esqueleto del sistema en su totalidad, y se le añaden los componentes. Esto se denomina *integración descendente*. De forma alternativa, pueden integrarse primero los componentes de infraestructura que proporcionan servicios comunes, tales como el acceso a bases de datos y redes, y a continuación pueden añadirse los componentes funcionales. Ésta es la *integración ascendente*. En la práctica, para muchos sistemas, la estrategia de integración es una mezcla de ambas, añadiendo en incrementos componentes de infraestructura y componentes funcionales. En ambas aproximaciones de integración, normalmente tiene que desarrollarse código adicional para simular otros componentes y permitir que el sistema se ejecute.

La principal dificultad que surge durante las pruebas de integración es la localización de los errores. Existen interacciones complejas entre los componentes del sistema, y cuando se descubre una salida anómala, puede resultar difícil identificar dónde ha ocurrido el error. Para hacer más fácil la localización de errores, siempre debería utilizarse una aproximación incremental para la integración y pruebas del sistema. Inicialmente, debería integrarse una configuración del sistema mínima y probar este sistema. A continuación, deberían añadirse componentes a esta configuración mínima y probar después de añadir cada incremento.

En el ejemplo mostrado en la Figura 23.3, A, B, C y D son componentes, y desde T1 hasta T5 son conjuntos de pruebas relacionados de las características incorporadas al sistema. T1, T2 y T3 se ejecutan primero sobre un sistema formado por los componentes A y B (el sistema mínimo). Si tales componentes revelan defectos, éstos se corrigen. El componente C se

integra y T1, T2 y T3 se repiten para asegurar que no ha habido interacciones no esperadas con A y B. Si surgen problemas en estas pruebas, esto significa probablemente que son debidos a las interacciones con el nuevo componente. Se localiza el origen del problema, simplificando así la localización y reparación de defectos. El conjunto de pruebas T4 se ejecuta también sobre el sistema. Finalmente, el componente D se integra y se prueba utilizando las pruebas existentes y nuevas (T5).

Cuando se planifica la integración, tiene que decidirse el orden de integración de los componentes. En un proceso como XP, el cliente se implica en el proceso de desarrollo y decide qué funcionalidad debería incluirse en cada incremento del sistema. Por lo tanto, la integración del sistema está dirigida por las prioridades del cliente. En otras aproximaciones al desarrollo, cuando se integran componentes comerciales y componentes especialmente desarrollados, el cliente puede no estar implicado y el equipo de integración decide sobre las prioridades de la integración.

En tales casos, una buena práctica es integrar primero los componentes que implementan las funcionalidades más frecuentemente utilizadas. Esto significa que los componentes más utilizados recibirán la mayoría de las pruebas. Por ejemplo, en el sistema de librería LIBSYS, debería comenzarse integrando la facilidad de búsqueda para que, en un sistema mínimo, los usuarios puedan buscar los documentos que necesitan. A continuación, se debería añadir la funcionalidad para permitir a los usuarios descargar un documento, y después agregar progresivamente los componentes que implementan otras características del sistema.

Por supuesto, la realidad es raramente tan simple como este modelo sugiere. La implementación de las características puede estar repartida entre varios componentes. Para probar una nueva característica, pueden tener que integrarse varios componentes diferentes. Las pruebas pueden revelar errores en las interacciones entre estos componentes individuales y otras partes del sistema. La reparación de errores puede ser difícil debido a que un grupo de componentes que implementan la característica del sistema pueden tener que cambiarse. Además, la integración y prueba de un nuevo componente puede cambiar el patrón de las interacciones de componentes ya probados. Se pueden manifestar errores que no habían aparecido en las pruebas de la configuración más simple.

Estos problemas significan que, cuando se integra un nuevo incremento, es importante volver a ejecutar las pruebas para incrementos previos, así como las nuevas pruebas requeridas

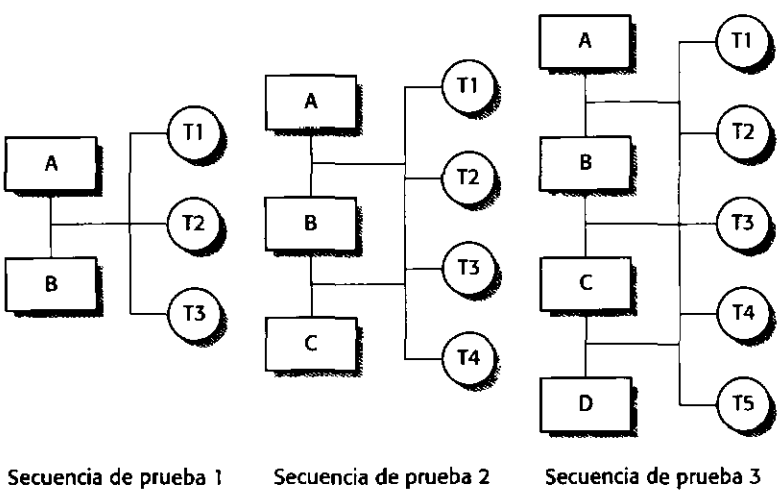


Figura 23.3 Pruebas de integración incrementales.

para verificar la nueva funcionalidad del sistema. Volver a ejecutar un conjunto existente de pruebas se denomina *pruebas de regresión*. Si las pruebas de regresión ponen de manifiesto problemas, entonces tiene que comprobarse si éstos son problemas en el incremento previo que el nuevo incremento ha puesto de manifiesto o si éstos son debidos al incremento añadido de funcionalidad.

Las pruebas de regresión son claramente un proceso caro y no resultan prácticas sin algún soporte automatizado. En programación extrema, tal y como se vio en el Capítulo 17, todas las pruebas se escriben como código ejecutable en donde la entrada de las pruebas y las salidas esperadas son especificadas y automáticamente comprobadas. Cuando esto se usa en un marco de trabajo de pruebas automatizado como JUnit (Massol y Husted, 2003), esto significa que las pruebas pueden volverse a ejecutar automáticamente. Es un principio básico de la programación extrema que el conjunto completo de pruebas se ejecute siempre que se integre nuevo código y que este nuevo código no sea aceptado hasta que todas las pruebas se ejecuten con éxito.

23.1.2 Pruebas de entregas

Las pruebas de entregas son el proceso de probar una entrega del sistema que será distribuida a los clientes. El principal objetivo de este proceso es incrementar la confianza del suministrador en que el sistema satisface sus requerimientos. Si es así, éste puede entregarse como un producto o ser entregado al cliente. Para demostrar que el sistema satisface sus requerimientos, tiene que mostrarse que éste entrega la funcionalidad especificada, rendimiento y confiabilidad, y que no falla durante su uso normal.

Las pruebas de entregas son normalmente un proceso de pruebas de caja negra en las que las pruebas se derivan a partir de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede ser determinado estudiando sus entradas y sus salidas relacionadas. Otro nombre para esto es *pruebas funcionales*, debido a que al probador sólo le interesa la funcionalidad y no la implementación del software.

La Figura 23.4 ilustra el modelo de un sistema que se admite en las pruebas de caja negra. El probador presenta las entradas al componente o al sistema y examina las correspondientes

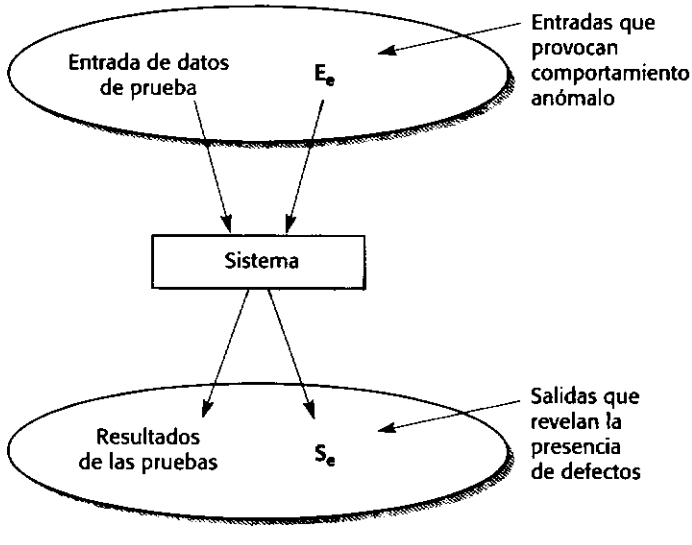


Figura 23.4 Pruebas de caja negra.

salidas. Si las salidas no son las esperadas (es decir, si las salidas pertenecen al conjunto S_e), entonces la prueba ha detectado un problema con el software.

Cuando se prueban las entregas del sistema, debería intentarse «romper» el software eligiendo casos de prueba que pertenecen al conjunto E_e en la Figura 23.4. Es decir, el objetivo debería ser seleccionar entradas que tienen una alta probabilidad de generar fallos de ejecución del sistema (salidas del conjunto S_e). Se utiliza la experiencia previa de cuáles son las pruebas de defectos que probablemente tendrán éxito y las guías de pruebas ayudarán a elegir la adecuada.

Autores adecuada como Whittaker (Whittaker, 2002) han recogido su experiencia de pruebas en un conjunto de guías que incrementan la probabilidad de que las pruebas de defectos tengan éxito. Algunos ejemplos de estas guías son:

1. Elegir entradas que fuerzan a que el sistema genere todos los mensajes de error.
2. Diseñar entradas que hacen que los búferes de entrada se desborden.
3. Repetir la misma entrada o series de entradas varias veces.
4. Forzar a que se generen las salidas inválidas.
5. Forzar los resultados de los cálculos para que sean demasiado grandes o demasiado pequeños.

Para validar que el sistema satisface los requerimientos, la mejor aproximación a utilizar es la prueba basada en escenarios, en la que se idean varios escenarios y se desarrollan casos de prueba a partir de estos escenarios. Por ejemplo, el siguiente escenario podría describir cómo el sistema de librería LIBSYS, tratado en capítulos anteriores, podría utilizarse:

Una estudiante escocesa que estudia la Historia Americana tiene que escribir un trabajo sobre «la mentalidad sobre las fronteras en el Este Americano desde 1840 a 1880». Para hacer esto, necesita encontrar documentación de varias bibliotecas. Se registra en el sistema LIBSYS y utiliza la facilidad de búsqueda para ver si puede acceder a los documentos originales de esa época. Descubre trabajos en varias bibliotecas universitarias de Estados Unidos y descarga copias de algunos de ellos. Sin embargo, para uno de los documentos, necesita tener confirmación de su Universidad de que ella es en verdad estudiante y de que el uso del documento es para fines no comerciales. La estudiante entonces utiliza la facilidad de LIBSYS que le permite solicitar dicho permiso y registrar su petición. Si ésta es aceptada, el documento podrá descargarse en el servidor de la biblioteca registrada y ser impreso. La estudiante recibe un mensaje de LIBSYS informándole que recibirá un mensaje de correo electrónico cuando el documento impreso esté disponible para ser recogido.

A partir de este escenario, es posible generar varias pruebas que pueden aplicarse a la entrega propuesta de LIBSYS:

1. Probar el mecanismo de login usando logins correctos e incorrectos para comprobar que los usuarios válidos son aceptados y que los inválidos son rechazados.
2. Probar la facilidad de búsqueda utilizando consultas con fuentes conocidas para comprobar que el mecanismo de búsqueda realmente encuentra los documentos.
3. Probar la facilidad de presentación del sistema para comprobar que la información sobre los documentos se visualiza adecuadamente.
4. Probar el mecanismo para solicitar permisos para descargas.
5. Probar la respuesta de correo electrónico indicando que el documento descargado está disponible.

Para cada una de estas pruebas, debería diseñarse un conjunto de pruebas que incluyan entradas válidas e inválidas y que generen salidas válidas e inválidas. También deberían organizarse pruebas basadas en escenarios para que los escenarios más probables sean probados primero, y los escenarios inusuales o excepcionales sean probados más tarde, de forma que el esfuerzo se centre en aquellas partes del sistema que reciben un mayor uso.

Si se han utilizado casos de uso para describir los requerimientos del sistema, estos casos de uso y los diagramas de secuencia asociados pueden ser una base para las pruebas del sistema. Los casos de uso y los diagramas de secuencia pueden emplearse ambos durante la integración y pruebas de entregas. Para ilustrar esto, se utiliza un ejemplo del sistema de estación meteorológica descrito en el Capítulo 14.

La Figura 23.5 muestra la secuencia de operaciones en la estación meteorológica cuando responde a una petición para recoger datos del sistema de mapas. Puede utilizarse este diagrama para identificar operaciones que serán probadas y ayudar al diseño de los casos de prueba para ejecutar las pruebas. Por lo tanto, la emisión de una petición de un informe dará lugar la ejecución de la siguiente secuencia de métodos:

CommsController:request → WeatherStation:report → WeatherData:summarise

El diagrama de secuencias también puede utilizarse para identificar entradas y salidas que tienen que crearse para las pruebas:

1. Una entrada de una petición de un informe debería tener un reconocimiento asociado y se debería devolver en última instancia un informe a partir de la petición. Durante las pruebas, se deberían crear datos resumidos que puedan utilizarse para comprobar que el informe está organizado correctamente.
2. Una petición de entrada de un informe a **WeatherStation** provoca la generación de un informe resumido. Se puede probar esto de forma aislada creando datos correspondientes al resumen que se ha preparado para las pruebas de **CommsController** y comprobar que el objeto **WeatherStation** genera correctamente este resumen.
3. Estos datos también se utilizan para probar el objeto **WeatherStation**.

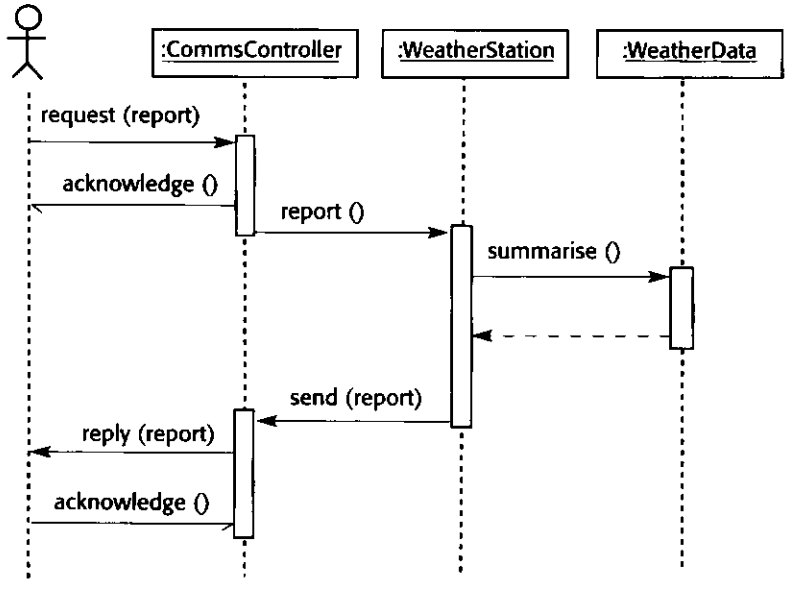


Figura 23.5 Diagrama de secuencia de recopilación de datos sobre el tiempo.

Por supuesto, se ha simplificado el diagrama de secuencia en la Figura 23.5 para que no muestre las excepciones. Un escenario completo de pruebas debería tener también éstas en cuenta y asegurar que los objetos manejan correctamente las excepciones.

23.1.3 Pruebas de rendimiento

Una vez que un sistema se ha integrado completamente, es posible probar las propiedades emergentes del sistema (véase el Capítulo 2) tales como rendimiento y fiabilidad. Las pruebas de rendimiento tienen que diseñarse para asegurar que el sistema pueda procesar su carga esperada. Esto normalmente implica planificar una serie de pruebas en las que la carga se va incrementando regularmente hasta que el rendimiento del sistema se hace inaceptable.

Como sucede con otros tipos de pruebas, las pruebas de rendimiento se ocupan tanto de demostrar que el sistema satisface sus requerimientos como de descubrir problemas y defectos en el sistema. Para probar si los requerimientos de rendimiento son alcanzados, usted tiene que construir un perfil operacional. Un perfil operacional es un conjunto de pruebas que reflejan la combinación real de trabajo que debería ser manejada por el sistema. Por lo tanto, si el 90% de las transacciones en un sistema son de tipo A, un 5% de tipo B y el resto de tipos C, D y E, entonces usted tiene que diseñar el perfil operacional para que la amplia mayoría de las pruebas sean de tipo A. En caso contrario, no se tendrá un test preciso del rendimiento operacional del sistema. Se analizan los perfiles operacionales y su uso en las pruebas de fiabilidad en el Capítulo 24.

Por supuesto, esta aproximación no es necesariamente la mejor aproximación para las pruebas de defectos. Tal y como se explica más adelante, la experiencia ha demostrado que una forma efectiva de descubrir defectos es diseñar pruebas alrededor de los límites del sistema. Las pruebas de rendimiento implican estresar el sistema (de ahí el nombre de pruebas de estrés) realizando demandas que están fuera de los límites del diseño del software.

Por ejemplo, un sistema de procesamiento de transacciones puede diseñarse para procesar hasta 300 transacciones por segundo; un sistema operativo puede diseñarse para gestionar hasta 1.000 terminales distintas. Las pruebas de estrés van realizando pruebas acercándose a la máxima carga del diseño del sistema hasta que el sistema falla. Este tipo de pruebas tienen dos funciones:

1. Prueba el comportamiento de fallo de ejecución del sistema. Pueden aparecer circunstancias a través de una combinación no esperada de eventos en donde la carga sobre el sistema supere la máxima carga anticipada. En estas circunstancias, es importante que un fallo de ejecución del sistema no provoque la corrupción de los datos o pérdidas inesperadas de servicios de los usuarios. Las pruebas de estrés verifican que las sobrecargas en el sistema provocan «fallos ligeros» en lugar de colapsarlo bajo su carga.
2. Sobrecargan el sistema y pueden provocar que se manifiesten defectos que normalmente no serían descubiertos. Aunque se puede argumentar que estos defectos es improbable que causen fallos de funcionamiento en un uso normal, puede haber combinaciones inusuales de circunstancias normales que las pruebas de estrés pueden reproducir.

Las pruebas de estrés son particularmente relevantes para los sistemas distribuidos basados en una red de procesadores. Estos sistemas exhiben a menudo una degradación grave cuando son sobrecargados. La red se satura con datos de coordinación que los diferentes procesos deben intercambiar, de forma que los procesos son cada vez más lentos a medida que esperan los datos requeridos de otros procesos.

23.2 Pruebas de componentes

Las pruebas de componentes (a menudo denominadas *pruebas de unidad*) son el proceso de probar los componentes individuales en el sistema. Éste es un proceso de pruebas de defectos, por lo que su objetivo es encontrar defectos en estos componentes. Tal y como se indicó en la introducción, para la mayoría de los sistemas, los desarrolladores de componentes son los responsables de las pruebas de componentes.

Existen diferentes tipos de componentes que pueden probarse en esta etapa:

1. Funciones individuales o métodos dentro de un objeto.
2. Clases de objetos que tienen varios atributos y métodos.
3. Componentes compuestos formados por diferentes objetos o funciones. Estos componentes compuestos tienen una interfaz definida que se utiliza para acceder a su funcionalidad.

Las funciones o métodos individuales son el tipo más simple de componente y sus pruebas son un conjunto de llamadas a estas rutinas con diferentes parámetros de entrada. Pueden utilizarse las aproximaciones para diseñar los casos de prueba, descritos en la sección siguiente, y para diseñar las pruebas de las funciones o métodos.

Cuando se están probando clases de objetos, deberían diseñar las pruebas para proporcionar cobertura para todas las características del objeto. Por lo tanto, las pruebas de clases de objetos deberían incluir:

1. Las pruebas aisladas de todas las operaciones asociadas con el objeto.
2. La asignación y consulta de todos los atributos asociados con el objeto.
3. Ejecutar el objeto en todos sus posibles estados. Esto significa que deben simularse todos los eventos que provocan un cambio de estado en el objeto.

Consideremos, por ejemplo, la estación meteorológica del Capítulo 14 cuya interfaz se muestra en la Figura 23.6. Ésta sólo tiene un único atributo, el cual es su identificador. Éste es una constante que se asigna cuando la estación meteorológica se instala. Por lo tanto, sólo se necesita una prueba que compruebe si dicho atributo ha sido actualizado. Se necesita definir casos de prueba para **reportWeather, calibrate, test, startup y shutdown.** En teoría, deberían probarse los métodos de forma independiente, pero, en algunos casos, son necesarias algunas secuencias de pruebas. Por ejemplo, para probar **shutdown** se necesita haber ejecutado el método **startup**.

Para probar los estados de la estación meteorológica, se utiliza un modelo de estados tal y como se muestra en la Figura 14.14. Mediante este modelo, se pueden identificar secuencias de transiciones de estados que tienen que ser probadas y definir secuencias de eventos para forzar estas transiciones. En principio, debería probarse cada posible secuen-

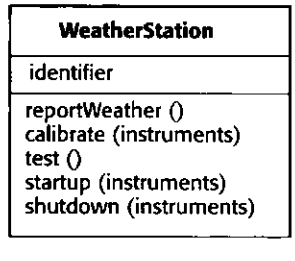


Figura 23.6
La interfaz del objeto de la estación meteorológica.

cia de transición de estados, aunque en la práctica esto puede suponer un coste demasiado elevado. Ejemplos de secuencias de estados que deberían probarse en la estación meteorológica son los siguientes:

Shutdown → Waiting → Shutdown
Waiting → Calibrating → Testing → Transmitting → Waiting
Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

Si se utiliza herencia, se hace más difícil diseñar las pruebas de clases de objetos. Siempre que una superclase proporcione operaciones que son heredadas por varias subclases, todas estas subclases deberían ser probadas con todas las operaciones heredadas. La razón de esto es que la operación heredada puede hacer suposiciones sobre otras operaciones y atributos, que pueden haber cambiado cuando se han heredado. Del mismo modo, cuando una operación de una superclase es sobrescrita, entonces la nueva operación debe ser probada.

La noción de clases de equivalencia, expuesta en la Sección 23.3.2, también puede aplicarse a clases de objetos. Las pruebas que pertenecen a la misma clase de equivalencia podrían ser aquellas que utilizan los mismos atributos de los objetos. Por lo tanto, deberían identificarse clases de equivalencia que inicializan, acceden y actualizan todos los atributos de las clases de objetos.

23.2.1 Pruebas de interfaces

Muchos componentes en un sistema no son simples funciones u objetos, sino que son componentes compuestos formados por varios objetos que interactúan. Tal y como se explicó en el Capítulo 19, que trataba la ingeniería del software basada en componentes, se accede a las funcionalidades de estos componentes a través de sus interfaces definidas. Entonces las pruebas de estos componentes se ocupan principalmente de probar que la interfaz del componente se comporta de acuerdo con su especificación.

La Figura 23.7 ilustra este proceso de pruebas de interfaces. Supongamos que los componentes A, B y C se han integrado para formar un componente más grande o subsistema. Los casos de prueba no se aplican a componentes individuales, sino a la interfaz del componente compuesto que se ha creado combinando estos componentes.

Las pruebas de interfaces son particularmente importantes para el desarrollo orientado a objetos y basado en componentes. Los objetos y componentes se definen por sus interfaces y

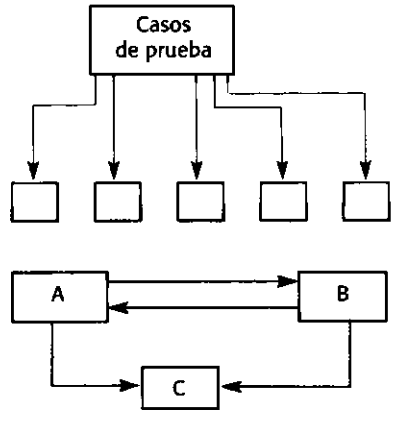


Figura 23.7 Pruebas de interfaces

pueden ser reutilizados en combinación con otros componentes en sistemas diferentes. Los errores de interfaz en el componente compuesto no pueden detectarse probando los objetos individuales o componentes. Los errores en el componente compuesto pueden surgir debido a interacciones entre sus partes.

Existen diferentes tipos de interfaces entre los componentes del programa y, consecuentemente, distintos tipos de errores de interfaces que pueden producirse:

1. *Interfaces de parámetros*. Son interfaces en las que los datos, o algunas veces referencias a funciones, se pasan de un componente a otro en forma de parámetros.
2. *Interfaces de memoria compartida*. Son interfaces en las que un bloque de memoria se comparte entre los componentes. Los datos se colocan en la memoria por un subsistema y son recuperados desde aquí por otros subsistemas.
3. *Interfaces procedurales*. Son interfaces en las que un componente encapsula un conjunto de procedimientos que pueden ser llamados por otros componentes. Los objetos y los componentes reutilizables tienen esta forma de interfaz.
4. *Interfaces de paso de mensajes*. Son interfaces en las que un componente solicita un servicio de otro componente mediante el paso de un mensaje. Un mensaje de retorno incluye los resultados de la ejecución del servicio. Algunos sistemas orientados a objetos tienen esta forma de interfaz, así como los sistemas cliente-servidor.

Los errores de interfaces son una de las formas más comunes de error en sistemas complejos (Lutz, 1993). Estos errores se clasifican en tres clases:

1. *Mal uso de la interfaz*. Un componente llama a otro componente y comete un error en la utilización de su interfaz. Este tipo de errores es particularmente común con interfaces de parámetros en donde los parámetros pueden ser de tipo erróneo, pueden pasarse en el orden equivocado o puede pasarse un número erróneo de parámetros.
2. *No comprensión de la interfaz*. El componente que realiza la llamada no comprende la especificación de la interfaz del componente al que llama, y hace suposiciones sobre el comportamiento del componente invocado. El componente invocado no se comporta como era de esperar y esto provoca un comportamiento inesperado en el componente que hace la llamada. Por ejemplo, puede llamarse a una rutina de búsqueda binaria con un vector no ordenado para realizar la búsqueda. En este caso, la búsqueda podría fallar.
3. *Errores temporales*. Se producen en sistemas de tiempo real que utilizan una memoria compartida o una interfaz de paso de mensajes. El productor de los datos y el consumidor de dichos datos pueden operar a diferentes velocidades. A menos que se tenga un cuidado particular en el diseño de la interfaz, el consumidor puede acceder a información no actualizada debido a que el productor de la información no ha actualizado la información de la interfaz compartida.

Las pruebas para encontrar defectos en las interfaces son difíciles debido a que algunos defectos de las interfaces sólo se pueden manifestar en condiciones inusuales. Por ejemplo, consideremos un objeto que implementa una cola con una estructura de datos de longitud fija. Un objeto que llama puede suponer que la cola está implementada como una estructura de datos infinita y puede no comprobar el desbordamiento de la cola cuando se introduce un elemento. Esta condición sólo se puede detectar durante las pruebas diseñando casos de prueba que fuerzan un desbordamiento de la cola y hacen que dicho desbordamiento no dañe el comportamiento del objeto de alguna forma detectable.

Puede surgir un problema adicional debido a las interacciones entre los defectos en distintos módulos u objetos. Los defectos en un objeto sólo pueden ser detectados cuando algún otro objeto se comporta de forma inesperada. Por ejemplo, un objeto puede llamar a algún otro objeto para recibir algún servicio y puede suponer que la respuesta es correcta. Si existe un malentendido sobre el valor calculado, el valor devuelto puede ser válido pero incorrecto. Esto sólo se manifestará cuando algún cálculo posterior sea erróneo.

He aquí algunas guías generales para las pruebas de interfaz:

1. Examinar el código a probar y listar explícitamente cada llamada a un componente externo. Diseñar un conjunto de pruebas en donde los valores de los parámetros para los componentes externos están en los extremos de sus rangos. Es bastante probable que estos valores extremos revelen inconsistencias en la interfaz.
2. En los lugares en los que se pasan punteros a través de una interfaz, siempre probar la interfaz con parámetros de punteros nulos.
3. Cuando se llama a un componente a través de una interfaz procedural, diseñar pruebas que hagan que el componente falle. Realizar suposiciones de fallos de ejecución erróneas es una de las malas interpretaciones de especificación más comunes.
4. Utilizar las pruebas de estrés, tal y como se indicó en la sección previa, en los sistemas de paso de mensajes. Diseñar pruebas que generen muchos más mensajes de los que probablemente ocurran en la práctica. Los problemas temporales se detectan de esta manera.
5. Cuando varios componentes interactúan a través de memoria compartida, diseñar pruebas que varían el orden en el que se activan estos componentes. Estas pruebas pueden revelar suposiciones implícitas hechas por el programador sobre el orden en el que los datos compartidos son producidos y consumidos.

Las técnicas de validación estáticas son a menudo más rentables que las pruebas para descubrir errores de interfaz. Un lenguaje fuertemente tipado como Java permite que muchos errores de interfaz sean detectados por el compilador. Si se utiliza un lenguaje débilmente tipado, tal como C, un analizador estático como LINT (véase el Capítulo 22) puede detectar errores de interfaz. Las inspecciones de programas se pueden centrar en las interfaces de los componentes y durante el proceso de inspección se pueden hacer preguntas sobre el comportamiento asumido de las interfaces.

23.3 Diseño de casos de prueba

El diseño de casos de prueba es una parte de las pruebas de componentes y sistemas en las que se diseñan los casos de prueba (entradas y salidas esperadas) para probar el sistema. El objetivo del proceso de diseño de casos de prueba es crear un conjunto de casos de prueba que sean efectivos descubriendo defectos en los programas y muestren que el sistema satisface sus requerimientos.

Para diseñar un caso de prueba, se selecciona una característica del sistema o componente que se está probando. A continuación, se selecciona un conjunto de entradas que ejecutan dicha característica, documenta las salidas esperadas o rangos de salida y, donde sea posible, se diseña una prueba automatizada que prueba que las salidas reales y esperadas son las mismas.

Existen varias aproximaciones que pueden seguirse para diseñar casos de prueba:

1. *Pruebas basadas en requerimientos*, en donde los casos de prueba se diseñan para probar los requerimientos del sistema. Esta aproximación se utiliza principalmente en la etapa de pruebas del sistema, ya que los requerimientos del sistema normalmente se implementan por varios componentes. Para cada requerimiento, se identifica casos de prueba que puedan demostrar que el sistema satisface ese requerimiento.
2. *Pruebas de particiones*, en donde se identifican particiones de entrada y salida y se diseñan pruebas para que el sistema ejecute entradas de todas las particiones y genere salidas en todas las particiones. Las particiones son grupos de datos que tienen características comunes, como todos los números negativos, todos los nombres con menos de 30 caracteres, todos los eventos provocados por la elección de opciones en un menú, y así sucesivamente.
3. *Pruebas estructurales*, en donde se utiliza el conocimiento de la estructura del programa para diseñar pruebas que ejecuten todas las partes del programa. Esencialmente, cuando se prueba un programa, debería intentarse ejecutar cada sentencia al menos una vez. Las pruebas estructurales ayudan a identificar casos de prueba que pueden hacer esto posible.

En general, cuando se diseñen casos de prueba, se debería comenzar con las pruebas de más alto nivel a partir de los requerimientos y a continuación, de forma progresiva, añadir pruebas más detalladas utilizando pruebas estructurales y pruebas de particiones.

23.3.1 Pruebas basadas en requerimientos

Un principio general de ingeniería de requerimientos, expuesto en el Capítulo 6, es que los requerimientos deberían poder probarse. Es decir, los requerimientos deberían ser escritos de tal forma que se pueda diseñar una prueba para que un observador pueda comprobar que los requerimientos se satisfacen. Las pruebas basadas en requerimientos, por lo tanto, son una aproximación sistemática al diseño de casos de prueba en donde el usuario considera cada requerimiento y deriva un conjunto de pruebas para cada uno de ellos. Las pruebas basadas en requerimientos son pruebas de validación en lugar de pruebas de defectos —el usuario intenta demostrar que el sistema ha implementado sus requerimientos de forma adecuada.

Por ejemplo, consideremos los requerimientos para el sistema LIBSYS introducidos en el Capítulo 6.

1. El usuario será capaz de buscar en un conjunto inicial de bases de datos o bien seleccionar un subconjunto de éstas.
2. El sistema proporcionará vistas apropiadas para que el usuario pueda leer los documentos almacenados.
3. Cada petición debería contener un único identificador (ORDER_ID) que el usuario deberá ser capaz de copiar en el área de peticiones de almacenamiento permanente.

Posibles pruebas para el primero de estos requerimientos, suponiendo que se ha probado una función de búsqueda, son:

- Iniciar búsquedas de usuario para elementos de los que se conoce que están presentes y para elementos que se sabe que no están presentes, en las que el conjunto de bases de datos incluye una base de datos.

- Iniciar búsquedas de usuario para elementos de los que se sabe que están presentes y para elementos de los que se sabe que no están presentes, en las que el conjunto de bases de datos incluye dos base de datos.
- Iniciar búsquedas de usuario para elementos de los que se sabe que están presentes y para elementos de los que se sabe que no están presentes, en las que el conjunto de bases de datos incluye más de dos base de datos.
- Seleccionar una base de datos del conjunto de bases de datos e iniciar búsquedas de usuario para elementos que se sabe que están presentes y para elementos de los que se sabe que no están presentes.
- Seleccionar más de una base de datos del conjunto de bases de datos e iniciar búsquedas de usuario para elementos de los que se sabe que están presentes y para elementos de los que se sabe que no están presentes.

Se puede ver a partir de esto que las pruebas de un requerimiento no significan escribir sólo una única prueba. Normalmente tienen que escribirse varias pruebas para asegurar que cubre por completo el requerimiento.

Las pruebas para los otros requerimientos en el sistema LIBSYS pueden desarrollarse de la misma forma. Para el segundo requerimiento, deberían escribirse pruebas para que pudieran ser procesados por el sistema documentos entregados de todos los tipos y comprobar que se visualizan adecuadamente. El tercer requerimiento es más sencillo. Para probarlo, se simula la emisión de varios pedidos y entonces se comprueba que el identificador del pedido está presente en la confirmación que recibe el usuario y que es única en cada caso.

23.3.2 Pruebas de particiones

Los datos de entrada y los resultados de salida de un programa normalmente se pueden agrupar en varias clases diferentes que tienen características comunes tales como números positivos, números negativos y selecciones de menús. Los programas normalmente se comportan de una forma similar para todos los miembros de una clase. Es decir, si se prueba un programa que realiza algún cálculo y requiere dos números positivos, entonces se esperaría que el programa se comportase de la misma forma para todos los números positivos.

Debido a este comportamiento equivalente, estas clases se denominan a menudo *particiones de equivalencia* o *dominios* (Bezier, 1990). Una aproximación sistemática al diseño de casos de prueba se basa en identificar todas las particiones para un sistema o componente. Los casos de prueba se diseñan para que las entradas o salidas pertenezcan a estas particiones. Las pruebas de particiones pueden utilizarse para diseñar casos de prueba tanto para sistemas como para componentes.

En la Figura 23.8, cada partición de equivalencia se muestra como una elipse. Las particiones de equivalencia son conjuntos de datos en donde todos los miembros de los conjuntos deberían ser procesados de forma equivalente. Las particiones de equivalencia de salida son resultados del programa que tienen características comunes, por lo que pueden considerarse como una clase diferente. También se identifican particiones en donde las entradas están fuera de otras particiones que se han elegido. Éstas prueban si el programa maneja entradas inválidas de forma correcta. Las entradas válidas e inválidas también forman particiones de equivalencia.

Una vez que se ha identificado un conjunto de particiones, pueden elegirse casos de prueba de cada una de estas particiones. Una buena práctica para la selección de casos de prueba es elegir casos de prueba en los límites de las particiones junto con casos de prueba cercanos

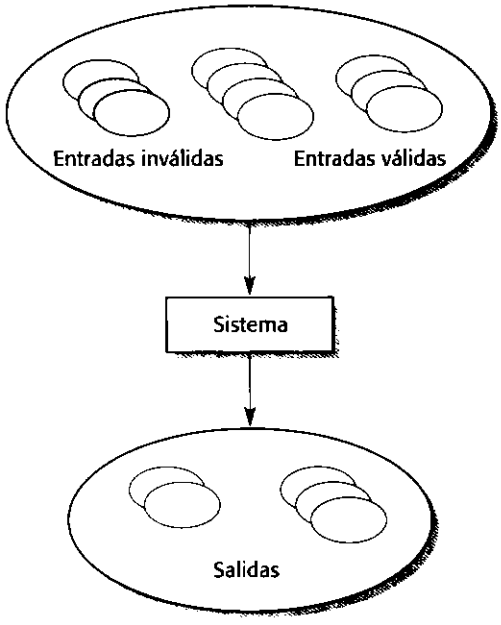


Figura 23.8
Particiones
de equivalencia.

al punto medio de la partición. La razón de esto es que los diseñadores y programadores tienden a considerar valores típicos de entradas cuando desarrollan un sistema. Éstos se prueban eligiendo el punto medio de la partición. Los valores límite son a menudo atípicos (por ejemplo, el cero puede comportarse de forma diferente del resto de los números no negativos), por lo que los diseñadores los pasan por alto. Los fallos de ejecución de los programas a menudo ocurren cuando se procesan estos valores atípicos.

Se identifican particiones usando la especificación del programa o documentación del usuario y, a partir de la propia experiencia, se predice qué clases de valores de entrada es probable que detecten errores. Por ejemplo, supongamos que una especificación de un programa indica que el programa acepta de 4 a 8 entradas que son enteros de cinco dígitos mayores de 10.000. La Figura 23.9 muestra las particiones para esta situación así como los posibles valores de prueba de entrada.

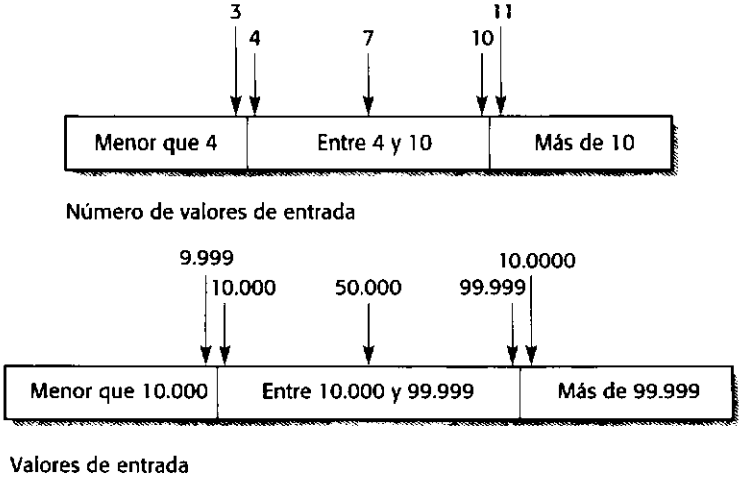


Figura 23.9
Ejemplos
de particiones
de equivalencia.

Para ilustrar la derivación de casos de prueba, se usa la especificación de un componente de búsqueda mostrado en la Figura 23.10. Este componente busca un elemento concreto (Key) en una secuencia de elementos. Devuelve la posición de dicho elemento en la secuencia. Se ha especificado esto de forma abstracta definiendo precondiciones, que son ciertas antes de que se llame al componente, y postcondiciones, que son ciertas después de su ejecución.

Las precondiciones indican que la rutina de búsqueda sólo funcionará con secuencias que no sean vacías. La postcondición indica que la variable Found toma un valor si el elemento buscado está en la secuencia. La posición del elemento buscado es el índice L. El valor del índice no está definido si el elemento no está en la secuencia.

A partir de esta especificación, pueden identificarse dos particiones de equivalencia:

1. Entradas en las que el elemento a buscar es un miembro de la secuencia (Found = true).
2. Entradas en las que el elemento a buscar no es un miembro de la secuencia (Found = false).

Cuando se están probando problemas con secuencias, vectores o listas, existen varias recomendaciones que a menudo son útiles para diseñar casos de prueba:

1. Probar el software con secuencias que tienen sólo un valor. Los programadores piensan de forma natural que las secuencias están formadas por varios valores, y algunas veces consideran esta suposición en sus programas. Como consecuencia, el programa puede no funcionar correctamente cuando se le presenta una secuencia con un único valor.
2. Utilizar varias secuencias de diferentes tamaños en distintas pruebas. Esto disminuye la probabilidad de que un programa con defectos produzca accidentalmente una salida correcta debido a alguna característica ocasional en la entrada.
3. Generar pruebas para acceder al primer elemento, al elemento central y al último elemento de la secuencia. Esta aproximación pone de manifiesto problemas en los límites de la partición.

A partir de estas recomendaciones, se pueden identificar dos particiones de equivalencia más:

1. La secuencia de entrada tiene un único valor.
2. El número de elementos de la secuencia de entrada es mayor que 1.

A continuación, se identifican particiones adicionales combinando estas particiones; por ejemplo, la partición en la que el número de elementos en la secuencia es mayor que 1 y el ele-

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
   -- la secuencia tiene al menos un elemento
   T'FIRST <= T'LAST
Post-condition
   -- el elemento se encuentra y es referenciado por L
   ( Found and T (L) = Key)
or
   -- el elemento no está en la secuencia
   ( not Found and
   not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

Figura 23.10 Especificación de una rutina de búsqueda.

Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Figura 23.11 Particiones de equivalencia para la rutina de búsqueda.

mento no pertenece a la secuencia. La Figura 23.11 muestra las particiones que se han identificado para probar el componente de búsqueda.

Un conjunto de posibles casos de prueba basados en estas particiones se muestran también en la Figura 23.11. Si el elemento a buscar no está en la secuencia, el valor de L no está definido («??»). La recomendación de que deberían utilizarse diferentes secuencias de distintos tamaños se ha aplicado en estos casos de prueba.

El conjunto de valores de entrada utilizados para probar la rutina de búsqueda no es exhaustivo. La rutina puede fallar si la secuencia de entrada incluye los elementos 1, 2, 3 y 4. Sin embargo, es razonable suponer que si la prueba falla al detectar defectos cuando uno de los miembros de la clase es procesado, ningún otro miembro de dicha clase identificará defectos. Por supuesto, los defectos todavía pueden existir. Algunas particiones de equivalencia pueden no haber sido identificadas, los errores pueden haberse cometido en la identificación de las particiones de equivalencia o los datos de las pruebas pueden no haberse preparado correctamente.

23.3.3 Pruebas estructurales

Las pruebas estructurales (Figura 23.12) son una aproximación al diseño de casos de prueba en donde las pruebas se derivan a partir del conocimiento de la estructura e implementación del software. Esta aproximación se denomina a veces pruebas de «caja blanca», de «caja de cristal» o de «caja transparente» para distinguirlas de las pruebas de caja negra.

La comprensión del algoritmo utilizado en un componente puede ayudar a identificar particiones adicionales y casos de prueba. Para ilustrar esto, se ha implementado la especifica-

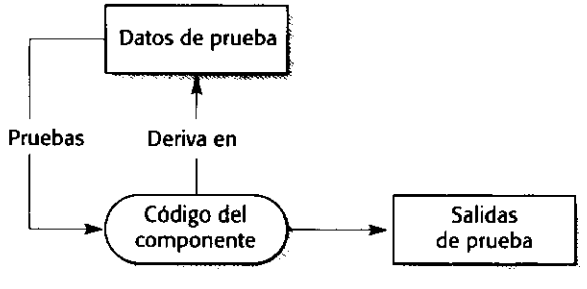


Figura 23.12 Pruebas estructurales.

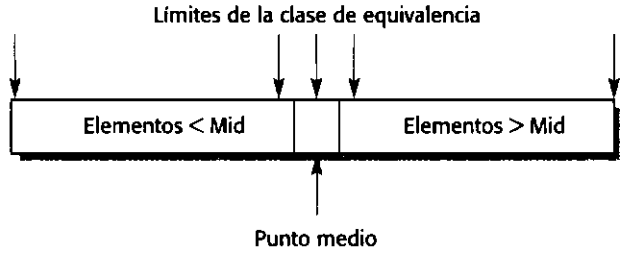


Figura 23.13 Clases de equivalencia de la búsqueda binaria.

ción de la rutina de búsqueda (Figura 23.10) como una rutina de búsqueda binaria (Figura 23.14). Por supuesto, ésta tiene precondiciones más estrictas. La secuencia se implementa como un vector, este vector debe estar ordenado y el valor del límite inferior del vector debe ser menor que el valor del límite superior.

Examinando el código de la rutina de búsqueda, puede verse que la búsqueda binaria implica dividir el espacio de búsqueda en tres partes. Cada una de estas partes constituye una partición de equivalencia (Figura 23.13). A continuación, se diseñan los casos de prueba en los que el elemento buscado se sitúa en los límites de cada una de estas particiones.

```
class BinSearch {

// Éste es un encapsulamiento de una función de búsqueda binaria que toma un
// vector de objetos ordenados y una clave y devuelve un objeto con 2 atributos:
// index – el valor del vector index
// found – un valor booleano que indica si key está en el vector.
// Se devuelve un objeto puesto que en Java no es posible pasar tipos básicos por
// referencia a una función y por lo tanto devolver dos valores.
// El valor de key es –1 si no se encuentra el elemento.

    public static void search ( int key, int [] elemArray, Result r )
    {
1.      int bottom = 0 ;
2.      int top = elemArray.length –1 ;
        int mid ;
3.      r.found = false ;
4.      r.index = –1 ;
5.      while ( bottom <= top )
        {
6.          mid = (top + bottom) / 2 ;
7.          if (elemArray [mid] == key)
            {
8.              r.index = mid ;
9.              r.found = true ;
10.             return ;
            } // if part
            else
            {
11.             if (elemArray [mid] < key)
12.                 bottom = mid + 1 ;
                else
13.                 top = mid –1 ;
            }
        } //while loop
14. } // búsqueda
} //BinSearch
```

Figura 23.14 Implementación Java de la rutina de búsqueda binaria.

17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Figura 23.15 Casos de prueba para la rutina de búsqueda.

Esto da lugar a un conjunto de casos de prueba revisados para la rutina de búsqueda, tal y como se muestra en la Figura 23.15. Observe que se ha modificado el vector de entrada para que esté ordenado de forma ascendente y se han añadido pruebas adicionales en las que el elemento a buscar es adyacente a la posición central del vector.

23.3.4 Pruebas de caminos

Las pruebas de caminos son una estrategia de pruebas estructurales cuyo objetivo es probar cada camino de ejecución independiente en un componente o programa. Si cada camino independiente, entonces todas las sentencias en el componente deben haberse ejecutado al menos una vez. Además, todas las sentencias condicionales comprueban para los casos verdadero y falso. En un proceso de desarrollo orientado a objetos, pueden utilizarse las pruebas de caminos cuando se prueban los métodos asociados a los objetos.

El número de caminos en un programa es normalmente proporcional a su tamaño. Puesto que los módulos se integran en sistemas, no es factible utilizar técnicas de pruebas estructurales. Por lo tanto, las técnicas de pruebas de caminos son principalmente utilizadas durante las pruebas de componentes.

Las pruebas de caminos no prueban todas las posibles combinaciones de todos los caminos en el programa. Para cualquier componente distinto de un componente trivial sin bucles, éste es un objetivo imposible. Existe un número infinito de posibles combinaciones de caminos en los programas con bucles. Incluso cuando todas las sentencias del programa se han ejecutado al menos una vez, los defectos del programa todavía pueden aparecer cuando se combinan determinados caminos.

El punto de partida de una prueba de caminos es un grafo de flujo del programa. Éste es un modelo del esqueleto de todos los caminos en el programa. Un grafo de flujo consiste en nodos que representan decisiones y aristas que muestran el flujo de control. El grafo de flujo se construye reemplazando las sentencias de control del programa por diagramas equivalentes. Si no hay sentencias goto en un programa, es un proceso sencillo derivar su grafo de flujo. Cada rama en una sentencia condicional (if-then-else o case) se muestra como un camino independiente. Una flecha que vuelve al nodo de la condición denota un bucle. Se ha dibujado el grafo de flujo para el método de búsqueda binaria en la Figura 23.16. Para establecer la correspondencia entre éste y el programa de la Figura 23.14 de forma más obvia, se ha mostrado cada sentencia como un nodo separado en el que cada número de nodo se corresponde con el mismo número de línea en el programa.

El objetivo de la prueba de caminos es asegurar que cada camino independiente en el programa se ejecuta al menos una vez. Un camino independiente del programa es aquel que recorre al menos una nueva arista en el grafo de flujo. En términos de programas, esto significa ejecutar una o más condiciones nuevas. Se deben ejecutar las ramas verdadera y falsa de todas las condiciones.

El grafo de flujo para el procedimiento de búsqueda binaria se muestra en la Figura 23.16, en donde cada nodo representa una línea en el programa con una sentencia ejecutable. Por lo tanto, realizando trazas del flujo se puede ver que los caminos en el grafo de flujo de búsqueda binaria son:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
1, 2, 3, 4, 5, 14
1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...

Si se ejecutan todos estos caminos, podemos estar seguros de que cada sentencia en el método ha sido ejecutada al menos una vez y que cada rama ha sido ejecutada para las condiciones verdadera y falsa.

Se puede encontrar el número de caminos independientes en un programa calculando la *complejidad ciclomática* (McCabe, 1976) del grafo de flujo del programa. Para programas sin sentencias goto, el valor de la complejidad ciclomática es uno más que el número de condiciones en el programa. Una condición simple es una expresión lógica sin conectores «and» u

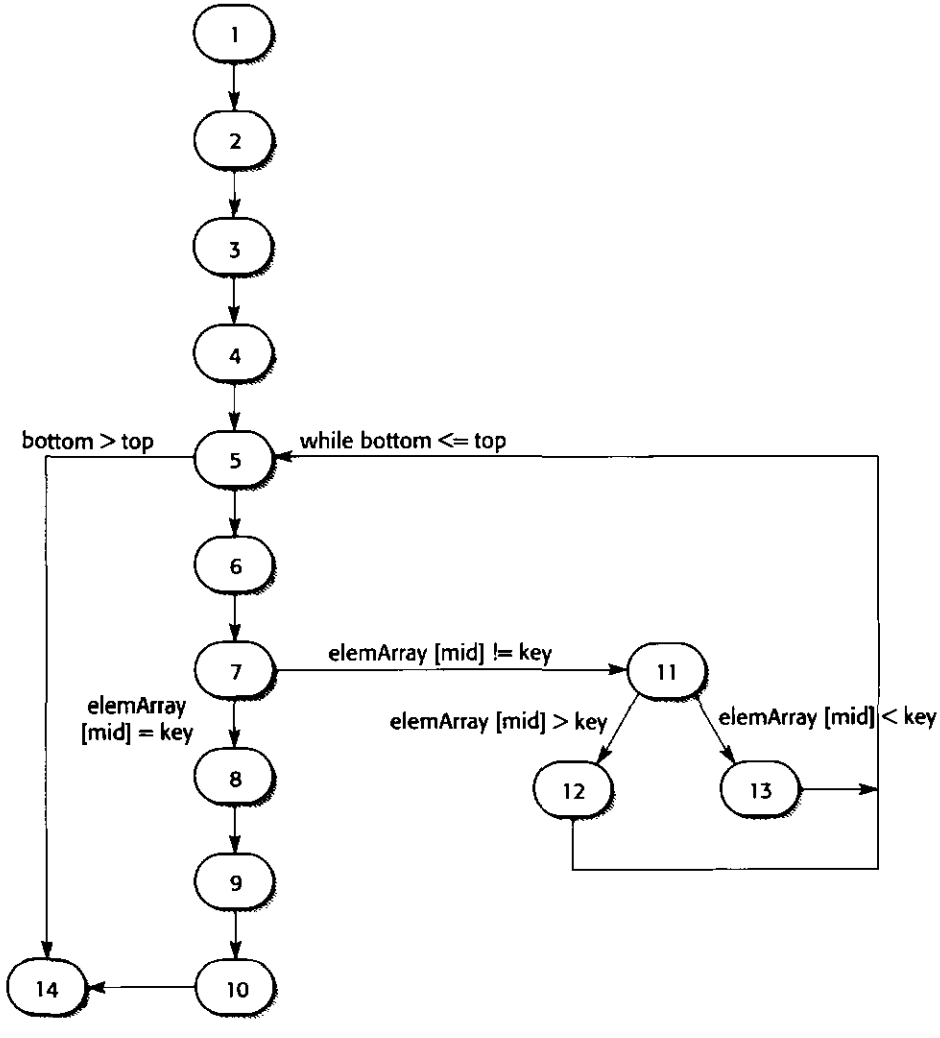


Figura 23.16
Grafo de flujo para una rutina de búsqueda binaria.

«or». Si el programa incluye condiciones compuestas, que son expresiones lógicas con conectores «and» u «or», entonces cuenta el número de condiciones simples en las condiciones compuestas cuando calcula la complejidad ciclomática.

Por lo tanto, si hay seis sentencias if y un bucle while y todas las expresiones condicionales son simples, la complejidad ciclomática es 8. Si una expresión condicional es una expresión compuesta tal como «if A and B or C», entonces esto se cuenta como tres condiciones simples. La complejidad ciclomática, por lo tanto, es 10. La complejidad ciclomática del algoritmo de búsqueda binaria (Figura 23.14) es 4 debido a que hay tres condiciones simples en las líneas 5, 7 y 11.

Después de descubrir el número de caminos independientes en el código calculando la complejidad ciclomática, se necesita diseñar casos de prueba para ejecutar cada uno de estos caminos. El número mínimo de casos de prueba necesarios para probar todos los caminos del programa es igual a la complejidad ciclomática.

El diseño de casos de prueba es sencillo en el caso de la rutina de búsqueda binaria. Sin embargo, cuando los programas tienen una estructura de ramas compleja, puede ser difícil predecir cómo deberá procesarse cualquier caso de prueba particular. En estos casos, para descubrir el perfil de ejecución del programa, puede utilizarse un analizador dinámico de programas.

Los analizadores dinámicos de programas son herramientas de pruebas que trabajan conjuntamente con los compiladores. Durante la compilación, estos analizadores añaden instrucciones adicionales al código generado. Éstos cuentan el número de veces que una sentencia ha sido ejecutada en un programa. Después de que el programa se ha ejecutado, puede imprimirse un perfil de ejecución. Éste muestra qué partes del programa han sido y no han sido ejecutadas utilizando casos de prueba particulares. Por lo tanto, este perfil de ejecución revela secciones del programa no probadas.

23.4 Automatización de las pruebas

Las pruebas son una fase cara y laboriosa del proceso del software. Como consecuencia, las herramientas de prueba estaban entre las primeras herramientas de software a desarrollar. Actualmente, estas herramientas ofrecen una serie de facilidades y su uso puede reducir significativamente los costes de las pruebas.

Ya se ha mostrado una aproximación para la automatización de las pruebas (Mosley y Posey, 2002) en las que se utiliza un marco de trabajo de pruebas tal como JUnit (Massol y Husted, 2003) para pruebas de regresión. JUnit es un conjunto de clases Java que el usuario extiende para crear un entorno de pruebas automatizado. Cada prueba individual se implementa como un objeto y un ejecutador de pruebas ejecuta todas las pruebas. Las pruebas en sí mismas deben escribirse de forma que indiquen si el sistema probado funciona como se esperaba.

Un banco de pruebas del software es un conjunto integrado de herramientas para soportar el proceso de pruebas. Además de a los marcos de trabajo de pruebas que soportan la ejecución automática de las pruebas, un banco de trabajo puede incluir herramientas para simular otras partes del sistema y generar datos de prueba de dicho sistema. La Figura 23.17 muestra algunas de las herramientas que podrían incluirse en un banco de trabajo de pruebas de este tipo:

1. *Gestor de pruebas*. Gestiona la ejecución de las pruebas del programa. El gestor de pruebas mantiene un registro de los datos de las pruebas, resultados esperados y faci-

lidades del programa que han sido probadas. Los marcos de trabajo automatizados tales como JUnit son ejemplos de gestores de pruebas.

2. *Generador de datos de prueba.* Genera datos de prueba para el programa a probar. Esto puede conseguirse seleccionando datos de una base de datos o utilizando patrones para generar datos aleatorios de forma correcta.
3. *Oráculo.* Genera predicciones de resultados esperados de pruebas. Los oráculos pueden ser versiones previas del programa o sistemas de prototipos. Las pruebas *back-to-back* (estudiadas en el Capítulo 17) implican ejecutar el oráculo y el programa a probar en paralelo. Las diferencias entre sus salidas son resaltadas.
4. *Comparador de ficheros.* Compara los resultados de las pruebas del programa con los resultados de pruebas previos e informa de las diferencias entre ellos. Los comparadores se utilizan en pruebas de regresión en las que se comparan los resultados de ejecutar diferentes versiones. Cuando se utilizan pruebas automatizadas, los comparadores pueden ser llamados desde las mismas pruebas.
5. *Generador de informes.* Proporciona la definición de informes y facilidades de generación para los resultados de las pruebas.
6. *Analizador dinámico.* Añade código a un programa para contar el número de veces que se ha ejecutado cada sentencia. Después de las pruebas, se genera un perfil de ejecución que muestra cuántas veces se ha ejecutado cada sentencia del programa.
7. *Simulador.* Se pueden utilizar diferentes tipos de simuladores. Los simuladores de la máquina objetivo simulan la máquina sobre la que se ejecuta el programa. Los simuladores de interfaces de usuario son programas conducidos por *scripts* que simulan múltiples interacciones de usuarios simultáneas. Utilizar simuladores para Entrada/Salida implica que el comportamiento temporal de la secuencia de las transacciones es repetible.

Cuando se utilizan para pruebas de grandes sistemas, las herramientas tienen que configurarse y adaptarse para el sistema específico que se está probando. Por ejemplo:

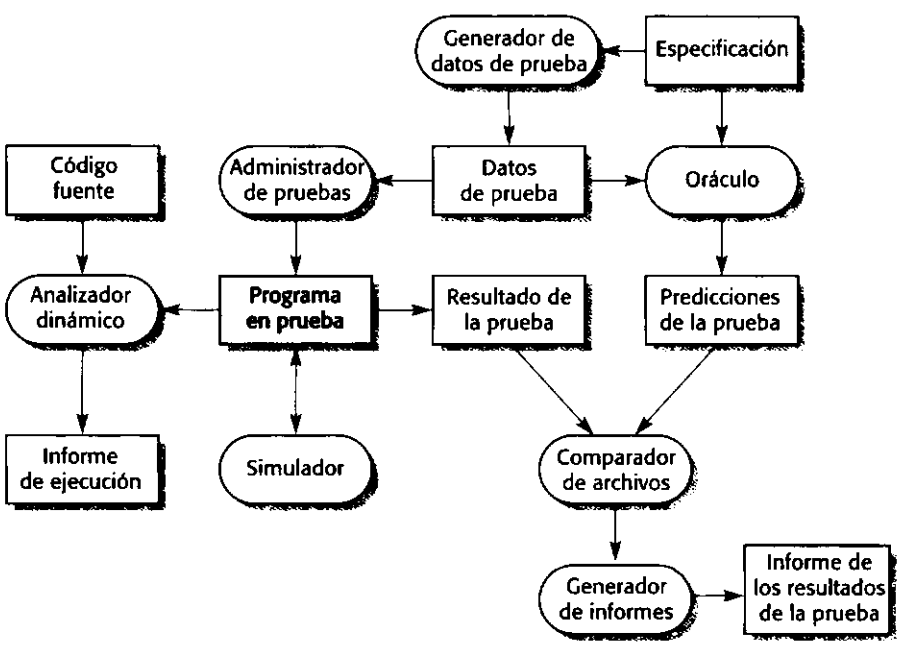


Figura 23.17
Un banco de trabajo de pruebas.

1. Pueden tener que añadirse nuevas herramientas para probar características específicas de la aplicación, y algunas herramientas existentes de prueba pueden no ser necesarias.
2. Pueden tener que escribirse scripts para simuladores de interfaz de usuario y definir patrones para generadores de datos de pruebas. Los formatos de los informes también pueden tener que ser definidos.
3. Pueden tener que prepararse manualmente los conjuntos de resultados esperados de las pruebas si no hay versiones previas de los programas disponibles que sirvan como oráculo.
4. Pueden tener que escribirse comparadores de ficheros de propósito especial que incluyan conocimiento de la estructura de los resultados de las pruebas sobre ficheros.

Normalmente, se necesita una cantidad significativa de esfuerzo y tiempo para crear un banco de trabajo de pruebas adecuado. Por lo tanto, los bancos de trabajo de pruebas completos, tal y como se muestra en la Figura 23.17, sólo se utilizan cuando se desarrollan sistemas grandes. Para estos sistemas, los costes totales de las pruebas pueden llegar al 50% del total de los costes de desarrollo, por lo que es rentable invertir en herramientas CASE de alta calidad para soportar las pruebas. Sin embargo, debido a que diferentes tipos de sistemas requieren distintos tipos de soportes para las pruebas, puede que no estén disponibles herramientas de pruebas comerciales. Rankin (Rankin, 2002) analiza una situación como ésta en IBM y describe el diseño del sistema de soporte de pruebas que desarrollaron para un servidor de comercio electrónico.

PUNTOS CLAVE

- **Las pruebas sólo pueden demostrar la presencia de errores en un programa. No pueden demostrar que no hay más defectos.**
- **Las pruebas de componentes son responsabilidad del desarrollador del componente. Un equipo independiente de pruebas lleva a cabo normalmente las pruebas del sistema.**
- **Las pruebas de integración son la actividad inicial de las pruebas del sistema en las que se prueban componentes integrados para detectar defectos. Las pruebas de entregas están relacionadas con las pruebas de las entregas al cliente y deberían validar que el sistema a entregar satisface sus requerimientos.**
- **Cuando se prueban los sistemas, debería intentarse «romper» el sistema usando la experiencia y recomendaciones para elegir los tipos de casos de prueba que han sido efectivos descubriendo defectos en otros sistemas.**
- **Las pruebas de interfaz intentan descubrir defectos en las interfaces de los componentes compuestos. Los defectos de las interfaces pueden ocurrir debido a errores cometidos en la lectura de la especificación, malentendidos en las especificaciones o errores o suposiciones temporales inválidas.**
- **Las particiones de equivalencia son una forma de derivar casos de prueba. Dependen de encontrar particiones en los conjuntos de datos de entrada y salida y ejecutar el programa con valores de estas particiones. A menudo, el valor que sea más probable que conduzca a una prueba con éxito es un valor en los límites de una partición.**

- **Las pruebas estructurales hacen referencia a analizar el programa para determinar caminos a través de él y usar este análisis como ayuda para la selección de los casos de prueba.**
- **La automatización de las pruebas reduce los costes de las pruebas apoyando al proceso de pruebas con varias herramientas software.**

LECTURAS ADICIONALES

How to Break Software: A Practical Guide to Testing. Este es un libro práctico más que teórico sobre las pruebas del software, en el que el autor presenta un conjunto de recomendaciones basadas en la experiencia sobre el diseño de las pruebas, que probablemente sean efectivas en el descubrimiento de defectos del sistema. (J. A. Whittaker, 2002, Addison-Wesley.)

«Software Testing and Verification». Este número especial del *IBM Systems Journal* contiene varios artículos sobre pruebas, incluyendo una buena revisión, artículos sobre métricas de pruebas y automatización de pruebas. [*IBM Systems Jounal*, 41(1), enero de 2002.]

Testing Object-oriented Systems: Models, Patterns and Tools. Este libro voluminoso proporciona un estudio completo sobre las pruebas orientadas a objetos. Su volumen indica que no debería ser el primer libro que se leyese sobre pruebas orientadas a objetos (la mayoría de los libros sobre desarrollo orientado a objetos tienen un capítulo de pruebas), pero claramente es el libro definitivo sobre pruebas orientadas a objetos. (R. V. Binder, 1999, Addison-Wesley.)

«How to design practical test cases». Un artículo sobre cómo diseñar casos de prueba por un autor de una compañía japonesa que tiene fama de entregar software con muy pocos defectos. [T. Yamaura, *IEEE Software*, 15(6), Noviembre 1998.]

EJERCICIOS

23.1 Explique por qué las pruebas sólo pueden detectar la presencia de errores, no su ausencia.

23.2 Compare una integración y pruebas ascendente y descendente comentando sus ventajas y desventajas para pruebas arquitectónicas, para mostrar una versión del sistema a los usuarios y para la implementación práctica y observación de las pruebas. Explique por qué la integración de la mayoría de los sistemas grandes, en la práctica, tiene que usar una mezcla de aproximaciones ascendentes y descendentes.

23.3 ¿Qué son las pruebas de regresión? Explique cómo el uso de pruebas automáticas y un marco de trabajo de pruebas tal como JUnit simplifica las pruebas de regresión.

23.4 Escriba un escenario que podría utilizarse como base para derivar pruebas del sistema de estación meteorológica que fue utilizado como ejemplo en el Capítulo 14.

23.5 Utilizando el diagrama de secuencia de la Figura 8.14 como escenario, proponga pruebas para la petición de elementos electrónicos en el sistema LIBSYS.

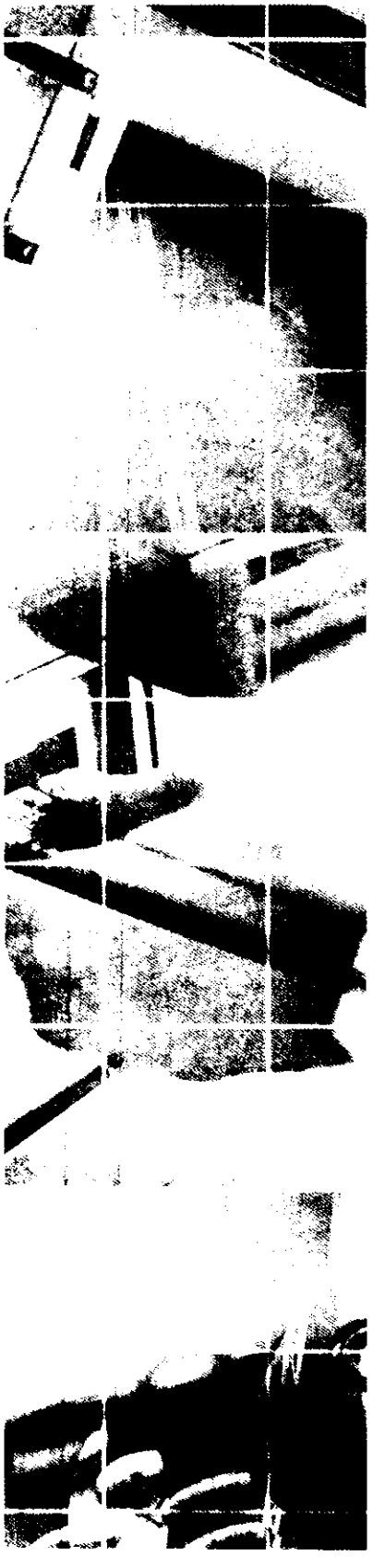
23.6 ¿Cuáles son los problemas que se plantean al desarrollar pruebas de rendimiento para un sistema de base de datos distribuida tal como el sistema LIBSYS?

23.7 Explique por qué las pruebas de interfaz son necesarias incluso cuando los componentes individuales han sido validados extensamente a través de las pruebas de componentes e inspecciones de programas.

23.8 Utilizando la aproximación presentada aquí para pruebas de objetos, diseñe casos de prueba para probar los estados del horno microondas cuyo modelo de estados se define en la Figura 8.5.

23.9 Se le ha solicitado que pruebe un método denominado catWhiteSpace en un objeto Paragraph que, dentro de un párrafo, reemplace secuencias de caracteres en blanco con un único carácter en blanco. Identifique particiones de pruebas para este ejemplo y derive un conjunto de pruebas para el método catWhiteSpace.

23.10 Indique tres situaciones en las que las pruebas de todos los caminos independientes en un programa pueden no detectar errores en el programa.



24 Validación de sistemas críticos

Objetivos

El objetivo de este capítulo es estudiar las técnicas de verificación y validación utilizadas en el desarrollo de sistemas críticos. Cuando haya leído este capítulo:

- comprenderá cómo puede medirse la fiabilidad del software y cómo los modelos de crecimiento de fiabilidad pueden utilizarse para predecir cuándo será alcanzado un nivel requerido de fiabilidad;
- comprenderá los principios de los argumentos de seguridad y cómo éstos pueden utilizarse junto con otros métodos de V & V para garantizar la seguridad de un sistema;
- comprenderá los problemas de garantizar la protección de un sistema;
- habrá sido introducido en casos de seguridad que presentan argumentos y evidencias de la seguridad de un sistema.

Contenidos

Obviamente, la verificación y validación de un sistema crítico tiene mucho en común con la validación de cualquier otro sistema. Los procesos de V & V deberían demostrar que el sistema satisface su especificación y que los servicios del sistema y su comportamiento están acordes con los requerimientos del cliente. Sin embargo, para sistemas críticos, en los que se requiere un alto nivel de confiabilidad, son necesarias pruebas y análisis adicionales para proporcionar la evidencia de que el sistema es confiable. Existen dos razones de por qué esto es necesario:

1. *Costes de fallos de ejecución.* Los costes y las consecuencias de los fallos de ejecución de los sistemas críticos son potencialmente mucho más grandes que para los sistemas no críticos. Pueden reducirse los riesgos de los fallos del sistema invirtiendo más en verificación y validación del sistema. Normalmente es más económico encontrar y eliminar defectos antes de que el sistema sea entregado que pagar por los consecuentes costes de accidentes o de un mal funcionamiento de los servicios del sistema.
2. *Validación de los atributos de confiabilidad.* Puede tenerse que hacer una demostración formal a los clientes de que el sistema satisface sus requerimientos especificados de confiabilidad (disponibilidad, fiabilidad, seguridad y protección). Para evaluar estas características de confiabilidad se requieren actividades específicas de V & V explicadas más adelante en este capítulo. En algunos casos, los reguladores externos, tales como autoridades de aviación nacionales, pueden tener que certificar que el sistema es seguro antes de que éste sea desplegado. Para obtener esta certificación, pueden tenerse que diseñar y llevar a cabo procedimientos de V & V especiales que recogen la evidencia sobre la confiabilidad del sistema.

Por estas razones, los costes de V & V para sistemas críticos son generalmente mucho mayores que para otras clases de sistemas. Es normal que el proceso de V & V consuma más del 50% de los costes totales de desarrollo para sistemas de software críticos. Por supuesto, este coste está justificado si se quiere evitar un fallo de ejecución del sistema que sea caro. Por ejemplo, en 1996 un sistema de software de misión crítica en el cohete Ariane 5 falló y se destruyeron varios satélites. Las pérdidas se cifraron en cientos de millones de dólares. La subsiguiente investigación descubrió que las deficiencias en el sistema de V & V fueron parcialmente responsables de este fallo.

Aunque el proceso de validación de sistemas críticos se centra principalmente en la validación del sistema, otras validaciones relacionadas deberían verificar que los procesos de desarrollo del sistema definidos han sido seguidos. Tal y como se explica en los Capítulos 27 y 28, la calidad del sistema se ve afectada por la calidad de los procesos utilizados para desarrollar el sistema. En resumen, buenos procesos conducen a buenos sistemas. Por lo tanto, para producir sistemas confiables, es necesario asegurarse de que se ha seguido un proceso de desarrollo robusto.

Esta garantía del proceso es una parte inherente de los estándares ISO 9000 para la gestión de la calidad, descritos brevemente en el Capítulo 27. Estos estándares requieren documentar los procesos que se utilizan y las actividades asociadas para asegurar que se han seguido estos procesos. Esto normalmente requiere la generación de registros del proceso, tales como formularios firmados, que certifiquen la finalización de las actividades del proceso y comprobaciones de calidad del producto. Los estándares ISO 9000 especifican qué salidas tangibles del proceso deberían producirse y quién es el responsable de producirlas. En la Sección 24.2.2 se proporciona un ejemplo de un registro para un proceso de análisis de contingencias.

24.1 Validación de la fiabilidad

Tal y como se explicó en el Capítulo 9, se han desarrollado varias métricas para especificar los requerimientos de fiabilidad de un sistema. Para validar que el sistema satisface estos requerimientos, tiene que medirse la fiabilidad del sistema tal y como lo ve un usuario típico del mismo.

El proceso de medir la fiabilidad de un sistema se ilustra en la Figura 24.1. Este proceso comprende cuatro etapas:

1. Se comienza estudiando los sistemas existentes del mismo tipo para establecer un perfil operacional. Un perfil operacional identifica las clases de entradas al sistema y la probabilidad de que estas entradas ocurran en un uso normal.
2. A continuación, se construye un conjunto de datos de prueba que reflejan el perfil operacional. Esto significa que se crean datos de prueba con la misma distribución de probabilidad que los datos de prueba para los sistemas que se han estudiado. Normalmente, se utiliza un generador de datos de prueba para soportar este proceso.
3. Se prueba el sistema utilizando estos datos y se contabiliza el número y tipo de fallos que ocurren. Los instantes en los que ocurren estos fallos también son registrados. Tal y como se indicó en el Capítulo 9, las unidades de tiempo que se elijan deberían ser adecuadas para la métrica de fiabilidad utilizada.
4. Después de que se ha observado un número de fallos significativos estadísticamente, se puede calcular la fiabilidad del software y obtener el valor adecuado de la métrica de fiabilidad.

Esta aproximación se denomina a menudo *pruebas estadísticas*. El objetivo de las pruebas estadísticas es evaluar la fiabilidad del sistema. Esto contrasta con la prueba de defectos, descrita en el Capítulo 23, en la que el objetivo es descubrir defectos del sistema. Prowell y otros (Prowell *et al.*, 1999) proporcionan una buena descripción de las pruebas estadísticas en su libro sobre ingeniería del software de Sala Limpia.

Esta aproximación para la medición de la fiabilidad es atractiva conceptualmente, pero no es fácil de aplicar en la práctica. Las principales dificultades que presenta son:

1. *Incertidumbre del perfil operacional.* Los perfiles operacionales basados en la experiencia con otros sistemas pueden no ser un reflejo exacto del uso real del sistema.
2. *Costes elevados de generación de datos de prueba.* Puede ser muy caro generar el gran volumen de datos requeridos en un perfil operacional a menos que el proceso pueda ser automatizado completamente.
3. *Incertidumbre estadística cuando se especifica una fiabilidad alta.* Se tiene que provocar un número de fallos significativo estadísticamente para permitir mediciones de fiabilidad exactas. Cuando el software ya es fiable, ocurren relativamente pocos fallos y es difícil provocar nuevos fallos.

Desarrollar un perfil operacional preciso es ciertamente posible para algunos tipos de sistemas, como los sistemas de telecomunicaciones, que tienen un patrón de uso estandarizado.

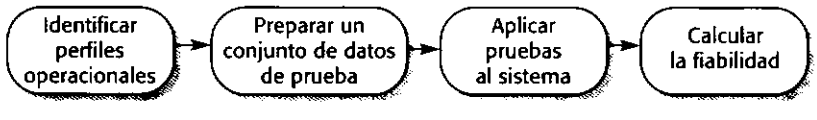


Figura 24.1 El proceso de medición de la fiabilidad.

Sin embargo, para otros tipos de sistemas, hay muchos usuarios diferentes que tienen su propia forma de utilizar el sistema. Tal y como se explicó en el Capítulo 3, distintos usuarios pueden obtener impresiones de fiabilidad completamente diferentes debido a que utilizan el sistema de forma distinta.

Con diferencia, la mejor forma de generar los grandes conjuntos de datos requeridos para la medición de la fiabilidad es utilizar un generador de datos de prueba que pueda generar automáticamente las entradas correspondientes al perfil operacional. Sin embargo, normalmente no es posible automatizar la producción de todos los datos de prueba para sistemas interactivos debido a que las entradas son a menudo una respuesta a las salidas del sistema. Los conjuntos de datos para estos sistemas tienen que generarse manualmente, con sus correspondientes costes más elevados. Incluso en los casos en los que es posible automatizar completamente el proceso, escribir los comandos para el generador de los datos de prueba puede llevar una cantidad de tiempo significativa.

La incertidumbre estadística es un problema general en la medición de la fiabilidad de un sistema. Para llevar a cabo predicciones precisas de fiabilidad, se necesita hacer algo más que simplemente provocar un único fallo de ejecución del sistema. Tiene que generarse un número razonablemente grande y significativo estadísticamente para tener la seguridad de que su medición de la fiabilidad es precisa. Cuanto más se disminuya el número de defectos en un sistema, más difícil resultará medir la efectividad de las técnicas de minimización de defectos. Si se especifican niveles muy altos de fiabilidad, a menudo no es práctico generar suficientes fallos del sistema para comprobar estas especificaciones.

24.1.1 Perfiles operacionales

El perfil operacional del software refleja cómo se utilizará éste en la práctica. Consiste en la especificación de clases de entradas y la probabilidad de su ocurrencia. Cuando un nuevo sistema software reemplaza a un sistema existente manual o automatizado, es razonablemente fácil evaluar el patrón de uso probable del nuevo software. Éste debería corresponderse con el uso del sistema existente, con algunas adiciones para las nuevas funcionalidades que (presumiblemente) se incluyen en el nuevo software. Por ejemplo, puede especificarse un perfil operacional para sistemas de centralitas de telecomunicaciones debido a que las compañías de telecomunicaciones conocen los patrones de llamadas que estos sistemas tienen que manejar.

Típicamente, el perfil operacional es tal que las entradas que tienen la probabilidad más alta de ser generadas se concentran en un pequeño número de clases, tal y como se muestra a la izquierda de la Figura 24.2. Hay un número extremadamente grande de clases en las que las entradas son altamente improbables, pero no imposibles. Éstas se muestran a la derecha de la Figura 24.2. Los puntos suspensivos (...) significan que existen más de estas entradas inusuales que no se muestran.

Musa (Musa, 1993; Musa, 1998) sugiere recomendaciones para el desarrollo de perfiles operacionales. Este autor trabajó en ingeniería de sistemas de telecomunicaciones, y existe una gran tradición en la recolección de datos de uso en este dominio. Como consecuencia, el proceso de desarrollo de perfiles operacionales es relativamente sencillo. Para un sistema que requiere alrededor de 15 personas-año de esfuerzo de desarrollo, se desarrolló un perfil operacional de alrededor de 1 persona-mes. En otros casos, el esfuerzo de la generación del perfil operacional fue mayor (2-3 personas-año), pero el coste disminuyó a lo largo de varias entregas del sistema. Musa se dio cuenta de que su compañía (una compañía

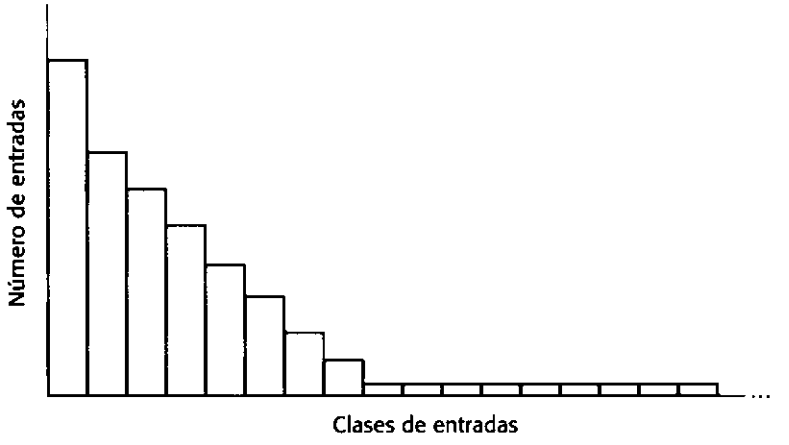


Figura 24.2
Un perfil operacional.

de telecomunicaciones) tuvo un beneficio de al menos diez veces la inversión requerida para desarrollar un perfil operacional.

Sin embargo, cuando un sistema software es nuevo e innovador, es difícil anticipar cómo será utilizado y, por lo tanto, generar un perfil operacional preciso. Muchos usuarios diferentes con distintas expectativas, conocimientos y experiencia pueden usar el nuevo sistema. No existen bases de datos históricas de uso. Estos usuarios pueden hacer uso de los sistemas en formas que no han sido anticipadas por los desarrolladores del sistema.

El problema se complica más debido a que los perfiles operacionales pueden cambiar conforme se utiliza el sistema. A medida que los usuarios comprenden el nuevo sistema y confían más en él, a menudo lo utilizan de forma más sofisticada. Debido a estas dificultades, Hamlet (Hamlet, 1992) sugiere que a veces es imposible desarrollar un perfil operacional fiable. Si el usuario no está seguro de que su perfil operacional es correcto, entonces no puede confiar en la exactitud de sus mediciones de fiabilidad.

24.1.2 Predicción de la fiabilidad

Durante la validación del software, los gestores tienen que dedicar esfuerzo a las pruebas del sistema. Puesto que el proceso de pruebas es muy caro, es importante dejar de probar tan pronto como sea posible y no «sobreprobar» el sistema. Las pruebas pueden detenerse cuando se alcance el nivel requerido de fiabilidad del sistema. Algunas veces, por supuesto, las predicciones de fiabilidad pueden revelar que el nivel requerido de fiabilidad nunca conseguirá. En este caso, el gestor debe tomar decisiones difíciles sobre la reescritura de parte del software o renegociar el contrato del sistema.

Un modelo de crecimiento de fiabilidad es un modelo de cómo cambia la fiabilidad del sistema a lo largo del tiempo durante el proceso de pruebas. A medida que se descubren los fallos del sistema, los defectos subyacentes que provocan estos fallos son reparados para que la fiabilidad del sistema mejore durante las pruebas y depuración. Para predecir la fiabilidad, el modelo conceptual de crecimiento de la fiabilidad debe ser traducido a un modelo matemático. Aquí no se entra en este nivel de detalle, sino que simplemente se plantea el principio del crecimiento de la fiabilidad.

Existen varios modelos de crecimiento de la fiabilidad que han sido derivados de experimentos de fiabilidad en varios dominios de aplicación diferentes. Tal y como Kan (Kan, 2003) pone de manifiesto, la mayoría de estos modelos son exponenciales, en los que la fia-

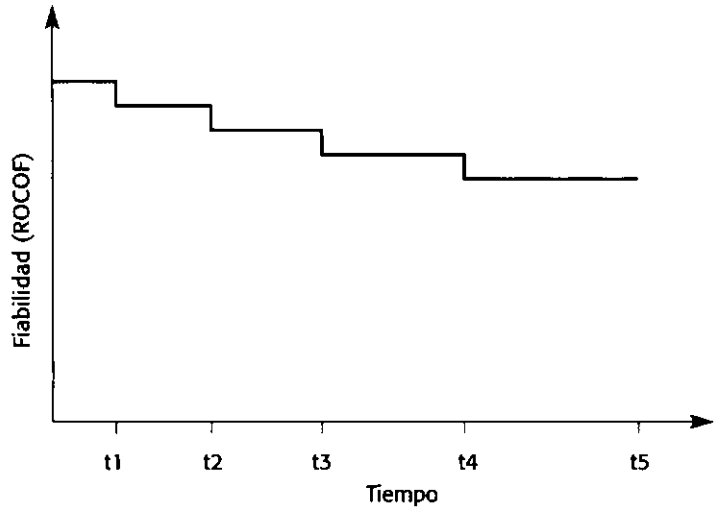


Figura 24.3 Modelo de función de pasos iguales del crecimiento de la fiabilidad.

bilidad crece rápidamente y los defectos son descubiertos y eliminados (véase la Figura 24.5). A continuación, el crecimiento se estabiliza y alcanza un nivel a medida que cada vez menos defectos son descubiertos y eliminados en posteriores etapas de pruebas.

El modelo más simple que ilustra el concepto de crecimiento de la fiabilidad es un modelo de funciones por pasos (Jelinski y Moranda, 1972). La fiabilidad crece de forma constante cada vez que un defecto (o un conjunto de defectos) es descubierto y reparado (Figura 24.3) y una nueva versión del software es creada. Este modelo supone que las reparaciones del software se implementan siempre correctamente, de forma que el número de defectos del software y fallos asociados decrece en cada nueva versión del sistema. A medida que tienen lugar las reparaciones, la tasa de ocurrencia de fallos del software (ROCOF) debería por tanto reducirse, tal y como se muestra en la Figura 24.3. Note que los periodos de tiempo sobre el eje horizontal reflejan el tiempo entre entregas del sistema para pruebas, de forma que normalmente tienen longitudes diferentes.

En la práctica, sin embargo, los defectos del software no siempre se reparan durante la depuración, y cuando se cambia un programa, a menudo se introducen nuevos defectos. La probabilidad de ocurrencia de estos defectos puede ser mayor que la probabilidad de ocurrencia del defecto que ha sido reparado. Por lo tanto, la fiabilidad del sistema a veces puede empeorar en una nueva entrega en lugar de mejorar.

El modelo simple de crecimiento de la fiabilidad de pasos iguales también supone que todos los defectos contribuyen de igual forma a la fiabilidad y que cada reparación de los defectos contribuye en la misma medida al crecimiento de la fiabilidad. Sin embargo, no todos los defectos son igualmente probables. Reparar los defectos más comunes contribuye más al crecimiento de la fiabilidad que reparar defectos que sólo ocurren de forma ocasional. Al usuario también le gustaría encontrar estos defectos probables cada vez en el proceso de pruebas, de forma que la fiabilidad puede crecer más que en etapas posteriores, en las que los defectos menos probables son descubiertos.

Modelos posteriores, como los sugeridos por Littlewood y Verrall (Littlewood y Verrall, 1973) tienen en cuenta estos problemas introduciendo un elemento aleatorio en la mejora del crecimiento de la fiabilidad conseguida por una reparación del software. Así, cada reparación no da como resultado una cantidad igual de la mejora de la fiabilidad, sino que varía dependiendo de la perturbación aleatoria (Figura 24.4).

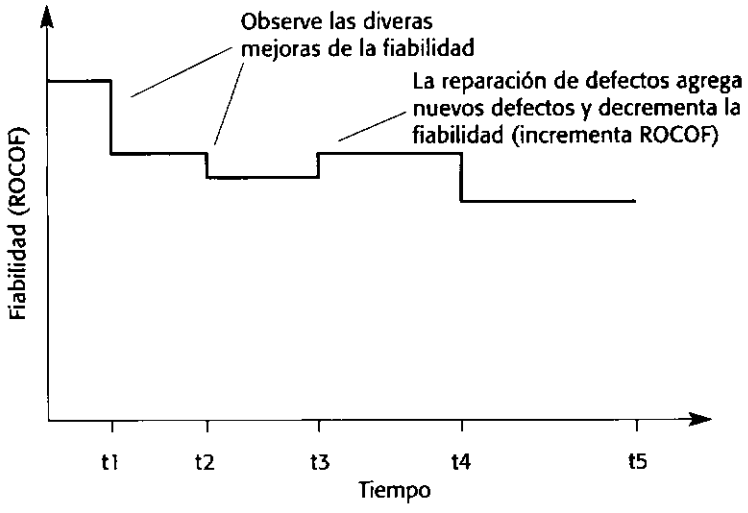


Figura 24.4
Modelo de función de pasos aleatorios del crecimiento de la fiabilidad.

El modelo de Littlewood y Verrall permite un crecimiento de la fiabilidad negativo cuando una reparación del software introduce errores adicionales. También modela el hecho de que a medida que los defectos son reparados, el promedio de mejora en cuanto a fiabilidad por reparación disminuye. La razón de esto es que los defectos más probables probablemente sean descubiertos pronto en el proceso de pruebas. La reparación de estos defectos contribuye más al crecimiento de la fiabilidad.

Los modelos anteriores son modelos discretos que reflejan el crecimiento de la fiabilidad de forma incremental. Cuando se entrega para las pruebas una nueva versión del software con defectos reparados debería haber una menor tasa de ocurrencia de fallos que en la versión previa. Sin embargo, para predecir la fiabilidad que deberá alcanzarse después de una determinada cantidad de pruebas, son necesarios modelos matemáticos continuos. Se han propuesto y comparado muchos modelos, derivados de diferentes dominios de aplicación (Littlewood, 1990).

De forma sencilla, puede predecirse la fiabilidad comparando los datos medidos de la fiabilidad con un modelo de fiabilidad conocido. A continuación, se extrapola el modelo al nivel requerido de fiabilidad y se observa cuándo se alcanzará dicho nivel (Figura 24.5). Por lo tanto, las pruebas y la depuración deben continuar hasta ese momento.

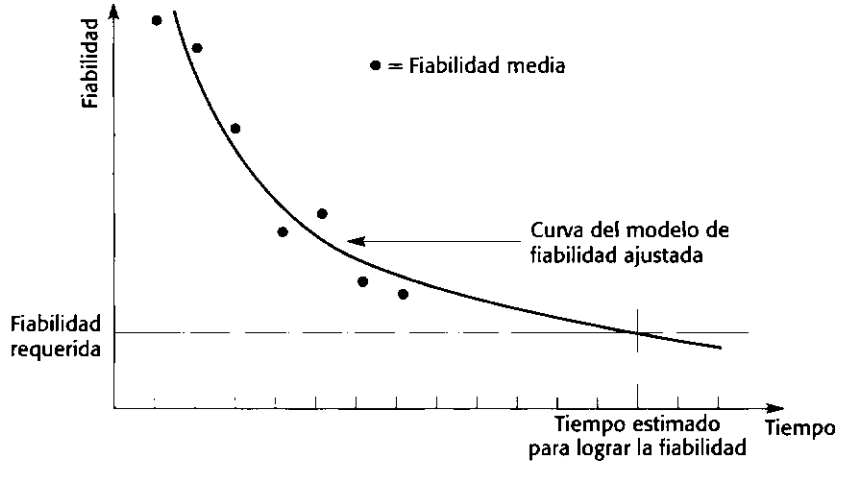


Figura 24.5
Predicción de la fiabilidad.

La predicción de la fiabilidad del sistema a partir de un modelo de crecimiento de fiabilidad tiene dos ventajas principales:

1. *Planificación de las pruebas.* Dado el calendario actual de pruebas, puede predecirse cuándo se completarán las pruebas. Si el final de las pruebas tiene lugar después de la fecha planificada de entrega del sistema, entonces puede tenerse que desplegar recursos adicionales para probar y depurar, y así acelerar la tasa de crecimiento de fiabilidad.
2. *Negociaciones con el cliente.* Algunas veces el modelo de fiabilidad muestra que el crecimiento de la fiabilidad es muy lento y que se requiere una cantidad de esfuerzo de pruebas desproporcionada para obtener un beneficio relativamente pequeño. Puede merecer la pena renegociar los requerimientos de fiabilidad con el cliente. De forma alternativa, puede ocurrir que el modelo prediga que la fiabilidad requerida probablemente nunca será alcanzada. En este caso, se tendrán que renegociar con el cliente los requerimientos de la fiabilidad del sistema.

Se ha simplificado aquí el modelado del crecimiento de la fiabilidad para proporcionarle un conocimiento básico del concepto. Si se desea utilizar estos modelos, se tiene que profundizar más y comprender las matemáticas subyacentes a estos modelos y sus problemas prácticos. Littlewood y Musa (Littlewood, 1990; Abdel-Ghaly *et al.*, 1986; Musa, 1998) han escrito extensamente sobre los modelos de crecimiento de la fiabilidad y Kan (Kan, 2003) tiene un excelente resumen en su libro. Varios autores han descrito su experiencia práctica en el uso de los modelos de crecimiento de fiabilidad (Ehrlich *et al.*, 1993; Schneidewind y Keller, 1992; Sheldon *et al.*, 1992).

24.2 Garantía de la seguridad

El proceso de la garantía de la seguridad y la validación de la fiabilidad tienen objetivos diferentes. Se puede especificar la fiabilidad de forma cuantitativa utilizando alguna métrica y a continuación medir la fiabilidad del sistema completo. Dentro de los límites del proceso de medición, se sabe si ha alcanzado el nivel requerido de fiabilidad. La seguridad, sin embargo, no puede especificarse de forma cuantitativa y, por lo tanto, no puede medirse cuando se prueba un sistema.

Por lo tanto, la garantía de la seguridad está relacionada con establecer un nivel de confianza en el sistema que podría variar desde «muy bajo» hasta «muy alto». Ésta es una cuestión de juicio profesional basado en evidencias sobre el sistema, su entorno y su proceso de desarrollo. En muchos casos, esta confianza está basada parcialmente en la experiencia de la organización que desarrolla el sistema. Si una compañía ha desarrollado previamente varios sistemas de control que funcionan de forma segura, entonces es razonable suponer que ésta continuará desarrollando sistemas seguros de este tipo.

Sin embargo, dicha evaluación debe contrastarse con evidencias tangibles a partir del diseño del sistema, los resultados de la V & V del sistema, y los procesos de desarrollo del sistema que se han utilizado. Para algunos sistemas, esta evidencia tangible se consigue en un caso de seguridad (véase la Sección 24.4) que permite a un regulador externo llegar a una conclusión justificada de la confianza del desarrollador en la seguridad del sistema.

Los procesos de V & V para sistemas de seguridad críticos tienen mucho en común con los procesos comparables de cualquier otro sistema con altos requerimientos de fiabilidad.

Se deben realizar unas pruebas generales para descubrir el mayor número posible de defectos, y cuando resulte apropiado, pueden utilizarse pruebas estadísticas para evaluar la fiabilidad del sistema. Sin embargo, debido a las tasas de fallos ultrabajas requeridas en muchos sistemas de seguridad críticos, las pruebas estadísticas no siempre proporcionan una estimación cuantitativa de la fiabilidad del sistema. Las pruebas simplemente proporcionan alguna evidencia, que se usa con alguna otra evidencia como los resultados de las revisiones y comprobaciones estáticas (véase el Capítulo 22), para hacer un juicio sobre la seguridad del sistema.

Las revisiones extensas son esenciales durante un proceso de desarrollo orientado a la seguridad, para exponer el software a la gente, que lo verá desde diferentes perspectivas. Parnas y otros (Parnas *et al.*, 1990) sugieren cinco tipos de revisiones que deberían ser obligatorias para los sistemas de seguridad críticos:

1. revisión para corregir la función que se pretende;
2. revisión para una estructura comprensible y mantenible;
3. revisión para verificar que el algoritmo y el diseño de las estructuras de datos son consistentes con el comportamiento especificado;
4. revisión de la consistencia del código y del diseño del algoritmo y de las estructuras de datos;
5. revisión de la adecuación de los casos de prueba del sistema.

Una suposición que subyace al trabajo en la seguridad de los sistemas es que el número de defectos en el sistema que puede dar lugar a contingencias de seguridad críticas es significativamente menor que el número total de defectos que pueden existir en el sistema. La garantía de la seguridad puede concentrarse en estos defectos con potencial de contingencia. Si puede demostrarse que estos defectos pueden no ocurrir o, si lo hacen, la contingencia asociada no provocará un accidente, entonces el sistema es seguro. Esta es la base de los argumentos de seguridad que se exponen en la siguiente sección.

24.2.1 Argumentos de seguridad

Las demostraciones de corrección de los programas, tal y como se se explicó en el Capítulo 22, han sido propuestas como una técnica de verificación del software desde hace más de treinta años. Las demostraciones formales de programas pueden ciertamente ser construidas para pequeños sistemas. Sin embargo, las dificultades prácticas para probar que un sistema satisface sus especificaciones son tan grandes que pocas organizaciones consideran las pruebas de corrección como uno de los costes. Sin embargo, para algunas aplicaciones críticas, puede ser rentable desarrollar pruebas de corrección para incrementar la confianza de que el sistema satisfaga sus requerimientos de seguridad o protección. En concreto, éste es el caso cuando la funcionalidad de seguridad crítica puede ser aislada en subsistemas muy pequeños que pueden ser especificados formalmente.

Aunque puede no ser rentable desarrollar demostraciones de corrección para la mayoría de los sistemas, a veces es posible desarrollar argumentos de seguridad simples que demuestran que el programa satisface sus obligaciones de seguridad. En un argumento de seguridad, no es necesario probar que la funcionalidad del programa es la que se especificó. Sólo es necesario demostrar que la ejecución del programa no conduce a un estado inseguro.

La técnica más efectiva para demostrar la seguridad de un sistema es la demostración por contradicción. Se comienza suponiendo que se está en un estado no seguro, el cual ha sido identificado por un análisis de contingencias del sistema, y que puede ser alcanzado ejecu-

```
-- The insulin dose to be delivered is a function of
-- blood sugar level, the previous dose delivered and
-- the time of delivery of the previous dose

    currentDose = computeInsulin () ;

    // Safety check--adjust currentDose if necessary

    // if-statement 1

    if (previousDose == 0)
    {
        if (currentDose > 16)
            currentDose = 16 ;
    }
    else
        if (currentDose > (previousDose * 2) )
            currentDose = previousDose * 2 ;

    // if-statement 2

    if ( currentDose < minimumDose )
            currentDose = 0 ;
    else if ( currentDose > maxDose )
            currentDose = maxDose ;
    administerInsulin (currentDose) ;
```

Figura 24.6 Código de suministro de insulina.

tando el programa. Se describe un predicado que define este estado no seguro. A continuación, se analiza el código de forma sistemática y se muestra que, para todos los caminos del programa que conducen a ese estado, la condición de terminación de estos caminos contradice el predicado no seguro. Si éste es el caso, la suposición inicial de un estado inseguro es incorrecta. Si se repite esto para todas las contingencias identificadas, entonces el software es seguro.

Como ejemplo, consideremos el código de la Figura 24.6, que podría ser parte de la implementación del sistema de suministro de insulina. Desarrollar un argumento de seguridad para este código implica demostrar que la dosis de insulina administrada nunca es mayor que algún nivel máximo establecido para cada individuo diabético. Por lo tanto, no es necesario probar que el sistema suministra la dosis correcta, sino simplemente que nunca suministra una sobredosis al paciente.

Para construir un argumento de seguridad, se identifica la precondición para el estado inseguro que, en este caso, es que currentDose > maxDose. Después se demuestra que todos los caminos del programa conducen a una contradicción de esta aserción no segura. Si éste es el caso, la condición no segura no puede ser cierta. Por lo tanto, el sistema es seguro. Se pueden estructurar y presentar los argumentos de seguridad de forma gráfica tal y como se muestra en la Figura 24.7.

Los argumentos de seguridad, según se refleja en la citada figura, son mucho más cortos que las verificaciones formales de sistemas. Primero se identifica todos los posibles caminos que conducen al estado potencialmente inseguro. Se trabaja hacia atrás a partir de este estado no seguro y se considera la última asignación de todas las variables de estado en cada camino que conduce a él. Pueden omitirse los cálculos previos (como la sentencia if 1 en la Figura 24.7) en el argumento de seguridad. En este ejemplo, todo lo que se necesita saber es el

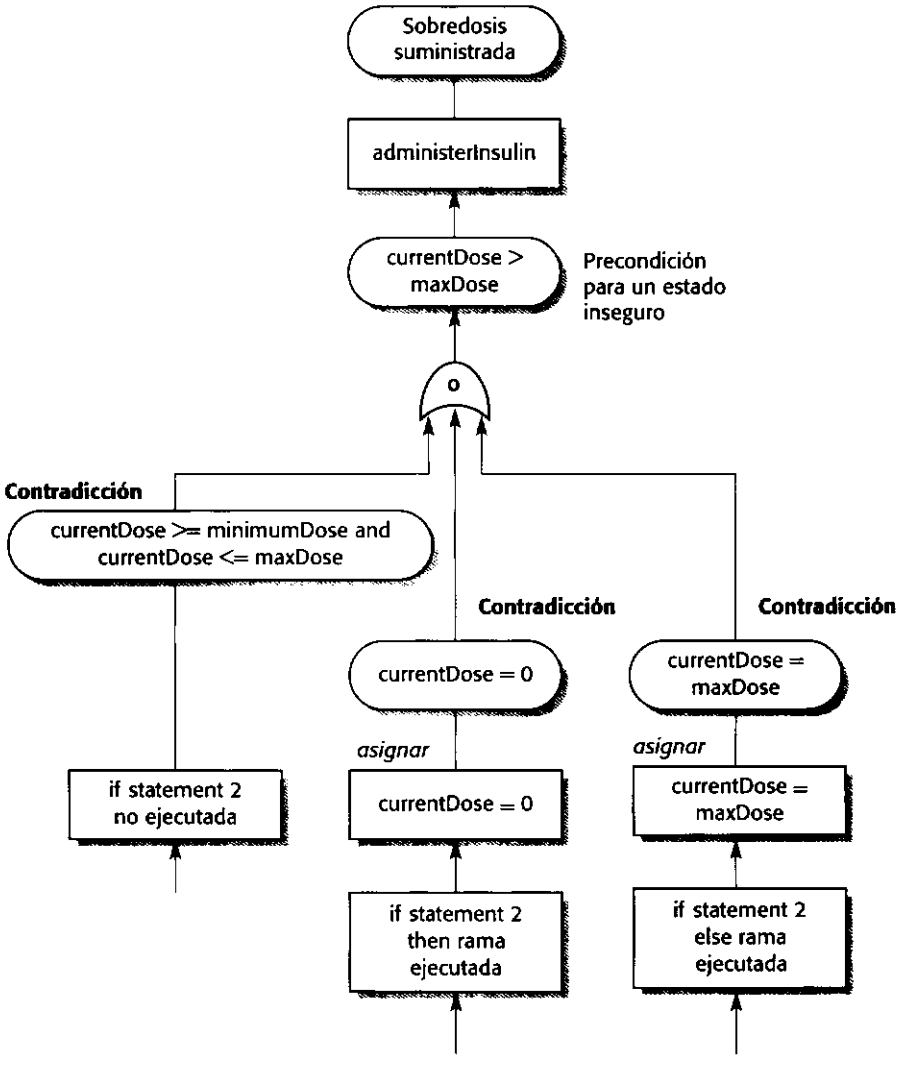


Figura 24.7
Argumento informal de seguridad basado en la demostración de contradicciones.

conjunto de posibles valores de **currentDose** inmediatamente antes de que el método **administerInsulin** sea ejecutado.

En el argumento de seguridad mostrado en la Figura 24.7, existen tres posibles caminos en el programa que conducen a la llamada al método **administerInsulin**. Queremos demostrar que la cantidad de insulina suministrada nunca excede del valor de **maxDose**. Se consideran todos los posibles caminos hasta **administerInsulin**:

1. Ninguna rama de la sentencia if 2 es ejecutada. Esto sólo puede ocurrir si **currentDose** es mayor o igual que **minimunDose** y menor o igual que **maxDose**.
2. La rama then de la sentencia if 2 es ejecutada. En este caso, se asigna el valor de cero a **currentDose**. Por lo tanto, su postcondición es **currentDose = 0**.
3. La rama else-if de la sentencia if 2 es ejecutada. En este caso, se asigna el valor **maxDose** a **currentDose**. Por lo tanto, su postcondición es **currentDose = maxDose**.

En los tres casos, las postcondiciones contradicen la precondición no segura de que la dosis administrada es mayor que **maxDose**, por lo que el sistema es seguro.

24.2.2 Garantía del proceso

En la introducción de este capítulo ya se ha puesto de manifiesto la importancia de garantizar la calidad del proceso de desarrollo del sistema. Esto es importante para todos los sistemas críticos, pero es particularmente importante para los sistemas de seguridad críticos. Existen dos razones de esto:

1. Los accidentes son eventos raros en los sistemas críticos y puede ser prácticamente imposible simularlos durante las pruebas de un sistema. No pueden realizarse pruebas extensivas para replicar las condiciones que conducen a un accidente.
2. Los requerimientos de seguridad, tal y como se ha visto en el Capítulo 9, son a menudo requerimientos «no debería» que excluyen comportamientos del sistema no seguros. Es imposible demostrar de forma concluyente a través de las pruebas y otras actividades de validación que estos requerimientos se han alcanzado.

El modelo de ciclo de vida para el desarrollo de sistemas de seguridad críticos (Capítulo 9, Figura 9.7) deja claro que debería prestarse atención a la seguridad durante todas las etapas del proceso del software. Esto significa que las actividades específicas de garantía de calidad deben incluirse en el proceso. Éstas incluyen:

1. La creación de un sistema de monitorización y registro de contingencias que siga una traza desde el análisis preliminar de contingencias hasta las pruebas y la validación del sistema.
2. La designación de los ingenieros de seguridad del proyecto que tienen responsabilidad explícita en aspectos de seguridad del sistema.
3. El uso frecuente de revisiones de seguridad durante todo el proceso de desarrollo.
4. La creación de un sistema de certificación de seguridad en el que la seguridad de los componentes críticos es certificada formalmente.
5. El uso de un sistema de gestión de configuraciones muy detallado (véase el Capítulo 29), que se utiliza para hacer un seguimiento de toda la documentación relacionada con la seguridad y tenerla a mano junto con la documentación técnica asociada. Es importante tener procesos de validación rigurosos si un fallo en la gestión de configuraciones implica que un sistema erróneo se entrega al cliente.

Para ilustrar la garantía de seguridad, se utiliza el proceso de análisis de contingencias que es una parte esencial del desarrollo de sistemas de seguridad críticos. El análisis de contingencias está relacionado con la identificación de contingencias, su probabilidad, y la probabilidad de que estas contingencias provoquen un accidente. Si el proceso de desarrollo incluye una clara trazabilidad desde la identificación de contingencias hasta el sistema mismo, entonces se puede argumentar por qué estas contingencias no provocan accidentes. Esto puede complementarse con argumentos de seguridad, tal y como se explicó en la Sección 24.2.1. Cuando se requiera una certificación externa antes de que el sistema sea utilizado (por ejemplo, en un avión), normalmente una condición de certificación es que la trazabilidad pueda ser demostrada.

El documento central de seguridad es el registro de contingencias, en el que se documentan y se lleva un seguimiento de las contingencias identificadas durante el proceso de especificación. A continuación, este registro de contingencias se utiliza en cada etapa del proceso de desarrollo del software para evaluar cómo esa etapa del desarrollo ha tenido en cuenta las contingencias. Un ejemplo simplificado de un registro de contingencias para el sistema de suministro de insulina se muestra en la Figura 24.8. Este formulario documenta el proceso de análisis de contingencias y muestra los requerimientos de diseño que han sido generados du-

Registro de contingencias **Página 4: Impresa 20.02.2003**
Sistema: Sistema de bomba de insulina *Archivo:* InsulinPump/Safety/HazardLog
Ingeniero de seguridad: James Brown *Versión del registro:* 1/3
Contingencia identificada Sobredosis de insulina suministrada al paciente
Identificada por Jane Williams
Clase de criticalidad 1
Riesgo identificado Alto
Árbol de defectos identificado Sí Fecha 24.01.99 *Lugar* Registro de contingencias, Página 5
Creadores del árbol de defectos Jane Williams y Bill Smith
Árbol de defectos verificado Sí Fecha 28.01.99 *Comprobador* James Brown

1. El sistema deberá incluir software de autoverificación que probará el sistema del sensor, el reloj y el sistema de insulina suministrada.
2. El software de autocomprobación deberá ejecutarse una vez por minuto.
3. En caso de que el software de autoverificación descubra un defecto en cualquiera de los componentes del sistema, se deberá emitir una alarma sonora y el despliegue de la bomba indicará el nombre del componente donde el defecto se descubrió. El suministro de insulina se suspenderá.
4. El sistema deberá incorporar un sistema de anulación que le permita al usuario del sistema modificar la dosis calculada de insulina que suministrará el sistema.
5. La cantidad a modificar no debe ser más grande que un valor prestablecido seleccionado cuando el sistema haya sido configurado por el personal médico.

Figura 24.8 Una página simplificada del registro de contingencias.

rante este proceso. Estos requerimientos de diseño intentan asegurar que el sistema de control nunca puede entregar una sobredosis de insulina a un usuario de la bomba de insulina.

Tal y como se muestra en la Figura 24.8, los individuos que tengan las responsabilidades en la seguridad deberían ser identificados explícitamente. Los proyectos de desarrollo de sistemas de seguridad críticos deberían tener siempre un ingeniero de seguridad del proyecto que no esté implicado en el desarrollo del sistema. La responsabilidad del ingeniero es asegurar que se hagan y documenten las comprobaciones adecuadas de seguridad. El sistema puede también requerir un asesor de seguridad independiente proveniente de una organización externa, que informe directamente al cliente sobre cuestiones de seguridad.

En algunos dominios, los ingenieros del sistema que tienen responsabilidades de seguridad deben ser certificados. En el Reino Unido, esto significa que tienen que haber sido aceptados como miembros de uno de los institutos de ingeniería (civil, eléctrico, mecánico, etc.) y tienen que ser ingenieros diplomados. Los ingenieros sin experiencia o poco cualificados no deben tener responsabilidades de seguridad.

Esto no se aplica actualmente a los ingenieros del software, aunque ha habido un gran debate sobre la concesión de licencias a ingenieros del software en varios estados de los Estados Unidos (Knight y Leveson, 2002; Bagert, 2002). Sin embargo, los futuros estándares de procesos para el desarrollo de software de seguridad crítico puede requerir que los ingenieros de seguridad del proyecto sean ingenieros certificados formalmente con un nivel de entrenamiento mínimo definido.

24.2.3 Comprobaciones de seguridad en tiempo de ejecución

Se ha descrito la programación defensiva en el Capítulo 20, en la cual se añaden sentencias redundantes a un programa para monitorizar su funcionamiento y comprobar posibles defec-

```
static void administerInsulin ( ) throws SafetyException {

    int maxIncrements = InsulinPump.maxDose / 8 ;
    int increments = InsulinPump.currentDose / 8 ;

    // assert currentDose <= InsulinPump.maxDose

    if (InsulinPump.currentDose > InsulinPump.maxDose)
        throw new SafetyException (Pump.doseHigh);
    else
        for (int i=1; i<= increments; i++)
        {
            generateSignal () ;
            if (i > maxIncrements)
                throw new SafetyException (Pump.incorrectIncrements);
        } // for loop

} //administerInsulin
```

Figura 24.9 Administración de insulina con comprobaciones en tiempo de ejecución.

tos en el sistema. La misma técnica puede utilizarse para monitorizar dinámicamente los sistemas de seguridad críticos. Puede añadirse código de comprobación del sistema que compruebe una restricción de seguridad. Éste lanza una excepción si se viola dicha restricción. Las restricciones de seguridad que deberían cumplirse siempre en puntos concretos de un programa pueden expresarse como *aserciones*. Las aserciones son predicados que describen condiciones que deberían cumplirse antes de que pueda ejecutarse la siguiente sentencia. En sistemas de seguridad críticos, las aserciones deberían generarse a partir de la especificación de seguridad. Las aserciones intentan asegurar el comportamiento seguro más que un comportamiento que esté de acuerdo con su especificación.

Las aserciones pueden ser particularmente valiosas para garantizar la seguridad de las comunicaciones entre los componentes del sistema. Por ejemplo, en el sistema de suministro de insulina, la dosis de insulina administrada implica generar señales a la bomba de insulina para suministrar un número específico de incrementos de insulina (Figura 24.9). El número de incrementos de insulina asociados con la dosis de insulina máxima permitida puede ser precalculado e incluido como una aserción en el sistema.

Si ha habido un error en el cálculo de currentDose, que es la variable de estado que almacena la cantidad de insulina a suministrar, o si este valor se ha dañado de alguna manera, entonces se detectará en este momento. No se suministrará una dosis excesiva de insulina, ya que la comprobación en el método asegura que la bomba no suministrará más de maxDose.

A partir de las aserciones de seguridad que están incluidas como comentarios en el programa, puede generarse código para comprobar estas aserciones. Puede verse esto en la Figura 24.9, en donde la sentencia if después del comentario de la aserción comprueba dicha aserción. En principio, mucha de esta generación de código puede ser automatizada utilizando un preprocesador de aserciones. Sin embargo, estas herramientas normalmente tienen que ser escritas de forma especial y el código de las aserciones se genera normalmente a mano.

24.3 Valoración de la protección

La valoración de la protección de un sistema está adquiriendo una importancia creciente ya que cada vez más sistemas críticos están conectados a Internet y pueden ser accedidos por

cualquiera que tenga una conexión de red. Diariamente hay historias sobre ataques a sistemas basados en web, y los virus y los gusanos se distribuyen normalmente a través de protocolos de Internet. Todo esto significa que los procesos de verificación y validación para sistemas basados en web deben centrarse en la evaluación de la protección, en la que se prueba la habilidad del sistema para resistir diferentes tipos de ataques; sin embargo, tal y como explica Anderson (Anderson, 2001), este tipo de evaluación de la seguridad es muy difícil de llevar a cabo. Como consecuencia, los sistemas a menudo son desplegados con agujeros de seguridad que los intrusos utilizan para conseguir el acceso o para dañar a estos sistemas.

Fundamentalmente, la razón de por qué la protección es tan difícil de evaluar, es que los requerimientos de protección, al igual que algunos requerimientos de seguridad, son requerimientos «no debería». Es decir, especifican qué es lo que no debería ocurrir en lugar de la funcionalidad del sistema o del comportamiento requerido. Normalmente no es posible definir este comportamiento no deseado como simples restricciones que pueden ser comprobadas por el sistema.

Si hay recursos disponibles, siempre puede demostrar que un sistema satisface sus requerimientos funcionales. Sin embargo, es imposible probar que un sistema no hace algo, por lo que, independientemente de la cantidad de pruebas, pueden quedar vulnerabilidades de protección en un sistema después de que éste haya sido desplegado. Incluso en los sistemas que han sido utilizados durante muchos años, un intruso ingenioso puede descubrir una nueva forma de atacar e introducirse en lo que se pensaba que era un sistema protegido. Por ejemplo, el algoritmo RSA para encriptación de datos que se pensó durante muchos años que era seguro, fue violado en 1999.

Existen cuatro aproximaciones complementarias para comprobar la protección:

1. *Validación basada en la experiencia.* En este caso, el sistema se analiza frente a tipos de ataques conocidos por el equipo de validación. Este tipo de validación se lleva a cabo normalmente junto con la validación basada en herramientas. Se pueden crear listas de comprobación de problemas de protección conocidos (Figura 24.10) para ayudar al proceso. Esta aproximación puede utilizar toda la documentación del sistema y podría ser parte de otras revisiones del sistema que comprueben errores u omisiones.
2. *Validación basada en herramientas.* En este caso, varias herramientas de protección, tales como comprobadores de contraseñas, se utilizan para analizar el sistema. Los com-

Lista de comprobaciones de seguridad

1. **¿Todos los ficheros creados por la aplicación tienen los permisos de acceso adecuados? Los permisos de acceso equivocados pueden llevar a que estos ficheros sean accedidos por usuarios no autorizados.**
2. **¿El sistema termina automáticamente las sesiones de usuario después de un periodo de inactividad? Las sesiones que se dejan activas pueden permitir accesos no autorizados a través de una computadora inesperada.**
3. **Si el sistema se ha escrito en un lenguaje de programación sin comprobación de límites de vectores, ¿existen situaciones en las que el desbordamiento del búfer pueda ser aprovechado? El desbordamiento de los búferes puede permitir a los intrusos enviar cadenas de código al sistema y a continuación ejecutarlas.**
4. **Si se establecen contraseñas, ¿comprueba el sistema que las contraseñas son «robustas»? Las contraseñas robustas consisten en mezclas de letras, números y signos de puntuación, y no son palabras normales de un diccionario. Son mucho más difíciles de violar que las contraseñas simples.**

Figura 24.10 Ejemplos de comprobaciones en una lista de comprobaciones de protección.

probadores de contraseñas detectan contraseñas inseguras tales como nombres comunes o cadenas de letras consecutivas. Ésta es realmente una extensión de la validación basada en la experiencia, en donde la experiencia se incluye en la herramienta usada.

3. *Equipos tigre.* En este caso, se forma un equipo y se le da el objetivo de romper la protección del sistema. Éstos simulan ataques al sistema y usan su ingenio para descubrir nuevas formas de comprometer la seguridad del sistema. Esta aproximación puede ser muy efectiva, especialmente si los miembros del equipo tienen experiencia previa en introducirse en los sistemas.
4. *Verificación formal.* Un sistema puede ser verificado frente a una especificación de protección formal. Sin embargo, al igual que en otras áreas, la verificación formal para la protección no se utiliza ampliamente.

Es muy difícil para los usuarios finales de un sistema verificar su protección. Como consecuencia, tal y como señala Gollmann (Gollmann, 1999), las organizaciones en Norteamérica y en Europa han establecido conjuntos de criterios de evaluación de protección que pueden ser comprobados por evaluadores especializados. Los proveedores de productos software pueden someter sus productos para su evaluación y certificación frente a estos criterios.

Por lo tanto, si se tiene un requerimiento para un nivel particular de protección, se puede elegir un producto que haya sido validado para ese nivel. Sin embargo, muchos productos no están certificados en cuanto a la protección o su certificación se aplica a productos individuales. Cuando el sistema certificado se utiliza junto con otros sistemas no certificados, como un software desarrollado localmente, entonces el nivel de protección del sistema completo no se puede evaluar.

24.4 Argumentos de confiabilidad y de seguridad

Los argumentos de seguridad y, más genéricamente, los argumentos de confiabilidad son documentos estructurados que proporcionan argumentos detallados y evidencias de que un sistema es seguro o de que se ha alcanzado un nivel requerido de confiabilidad. Para muchos tipos de sistemas críticos, la producción de un argumento de seguridad es un requerimiento legal, y el argumento debe satisfacer alguna certificación antes de que el sistema pueda ser desplegado.

Los reguladores son creados por los gobiernos para asegurar que las industrias privadas no se aprovechan de no seguir estándares nacionales para seguridad, protección, y así sucesivamente. Existen reguladores en muchas industrias diferentes. Por ejemplo, las líneas aéreas son reguladas por las autoridades de la aviación nacional tales como la FAA (en Estados Unidos) y la CAA (en el Reino Unido). Los reguladores de las líneas ferroviarias existen para asegurar la seguridad de las vías de tren, y los reguladores nucleares deben certificar la seguridad de una planta nuclear antes de que sea puesta en marcha. En el sector bancario, los bancos nacionales sirven como reguladores, estableciendo procedimientos y prácticas para reducir la probabilidad de fraude y proteger a los clientes de los bancos de las prácticas bancarias arriesgadas. A medida que los sistemas software son cada vez más importantes en la infraestructura crítica de los países, estos reguladores están cada vez más relacionados con los argumentos de seguridad y confiabilidad para sistemas software.

La función de un regulador es comprobar que un sistema finalizado es tan seguro como práctico, por lo que la figura del regulador se ve implicada principalmente cuando se ha completado el desarrollo del proyecto. Sin embargo, los reguladores y los desarrolladores rara-

mente trabajan de forma aislada; se comunican con el equipo de desarrollo para establecer qué tiene que incluirse en el argumento de seguridad. El regulador y los desarrolladores examinan conjuntamente los procesos y los procedimientos para asegurarse de que éstos están siendo establecidos y documentados para satisfacer al regulador.

Por supuesto, el software en sí mismo no es peligroso. Sólo cuando éste está embebido en un gran sistema socio-técnico o basado en computadora, los fallos de ejecución de dicho software pueden provocar fallos en otros equipos o procesos que a su vez pueden ocasionar lesiones o muertes. Por lo tanto, un argumento de seguridad del software siempre forma parte de un argumento de seguridad de un sistema más amplio que demuestra la seguridad del sistema completo. Cuando se construye un argumento de seguridad del software, se tienen que relacionar los fallos de ejecución del software con fallos del sistema más amplios y demostrar que estos fallos de ejecución no ocurrirán o que no se propagarán, de tal forma que puedan producirse fallos peligrosos del sistema.

Un argumento de seguridad es un conjunto de documentos que incluye una descripción del sistema, que tiene que ser certificada, más información sobre los procesos utilizados para desarrollar el sistema y, lo más crítico, argumentos lógicos que demuestran que el sistema es probablemente seguro. Más concretamente, Bishop y Bloomfield (Bishop y Bloomfield, 1998; Bishop y Bloomfield, 1995) definen un argumento de seguridad como:

> *Un conjunto de evidencias documentadas que proporciona un argumento válido y convincente de que un sistema es adecuadamente seguro para una aplicación determinada en un entorno concreto.*

La organización y contenidos de un argumento de seguridad depende del tipo de sistema que tiene que certificarse y su contexto de funcionamiento. La Figura 24.11 muestra una posible organización para un argumento de seguridad del software.

Descripción del sistema	**Una descripción del sistema y de sus componentes críticos.**
Requerimientos de seguridad	**Los requerimientos de seguridad abstraídos de la especificación de requerimientos del sistema.**
Análisis de riesgos y contingencias	**Documentos que describen las contingencias y riesgos que tienen que identificarse y las medidas que hay que tomar para reducir el riesgo .**
Análisis del diseño	**Conjunto de argumentos estructurados que justifican por qué el diseño es seguro.**
Verificación y validación	**Descripción de los procedimientos de V & V utilizados y, cuando sea adecuado, los planes de prueba del sistema.**
Informes de revisiones	**Informes de todos las revisiones de diseño y seguridad.**
Competencias del equipo	**Evidencia de la competencia de todos los equipos implicados en el desarrollo y validación de sistemas seguros.**
Procesos de garantía de calidad	**Informes de los procesos de garantía de calidad llevados a cabo durante el desarrollo del sistema.**
Procesos de gestión de cambios	**Informes de todos los cambios propuestos, acciones tomadas y, cuando sea apropiado, justificación de la seguridad de estos cambios.**
Argumentos de seguridad asociados	**Referencias a otros argumentos de seguridad que pueden influir en un argumento de seguridad.**

Figura 24.11 Componentes de un argumento de seguridad del software.

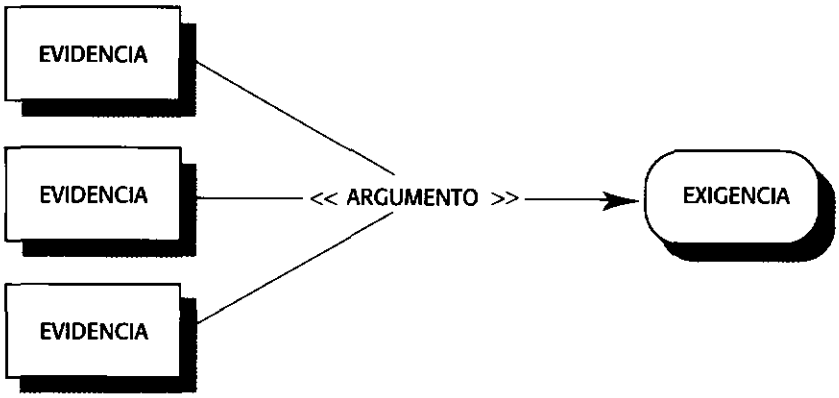


Figura 24.12 Estructura de un argumento.

Un componente clave de un argumento de seguridad es un conjunto de argumentos lógicos para la seguridad del sistema. Éstos pueden ser argumentos absolutos (el evento X ocurrirá o no) o argumentos probabilísticos (la probabilidad del evento X es 0.Y); al combinarlos, éstos deberían demostrar la seguridad. Tal y como se muestra en la Figura 24.12, un argumento es una relación entre lo que se piensa que debe ser un argumento (una exigencia) y un conjunto de evidencias que han sido observadas. El argumento esencialmente explica por qué la exigencia (que generalmente es que algo es seguro) puede inferirse a partir de la evidencia disponible. Naturalmente, dada la naturaleza multinivel de los sistemas, las exigencias se organizan en una jerarquía. Para demostrar que una exigencia de alto nivel es válida, primero se tiene que trabajar a través de los argumentos desde exigencias de niveles inferiores. La Figura 24.13 muestra una parte de esta jerarquía de exigencias para la bomba de insulina.

Como dispositivo médico, el sistema de bomba de insulina tiene que ser certificado externamente. Por ejemplo, en el Reino Unido, la Dirección de Dispositivos Médicos tiene que emitir un certificado de seguridad para cualquier dispositivo médico que se venda en el Reino

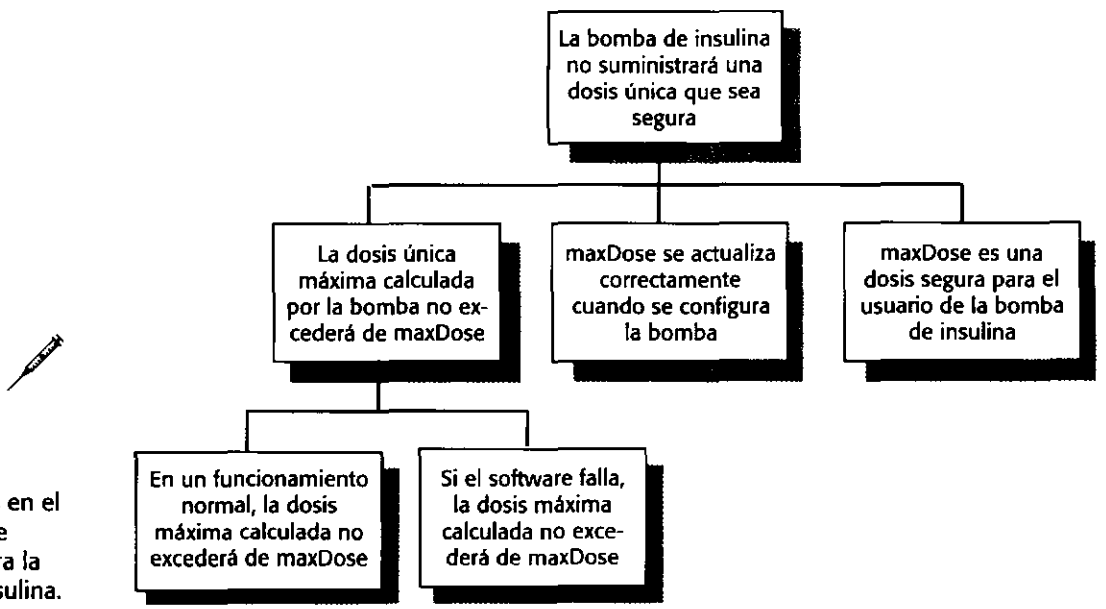


Figura 24.13 Jerarquía de exigencias en el argumento de seguridad para la bomba de insulina.

Unido. Pueden tener que generarse distintos argumentos para demostrar la seguridad de este sistema. Por ejemplo, el siguiente argumento podría formar parte de un argumento de seguridad para la bomba de insulina.

Exigencia: La dosis individual máxima de insulina calculada por la bomba no excederá de maxDose.

Evidencia: Argumento de seguridad para la bomba de insulina (Figura 24.7).

Evidencia: Conjuntos de datos de prueba para la bomba de insulina.

Evidencia: Informe de análisis estático para el software de la bomba de insulina.

Argumento: El argumento de seguridad presentado muestra que la dosis máxima de insulina que puede ser calculada es igual a maxDose.
En 400 pruebas, el valor de Dose fue calculado correctamente y nunca excedió de maxDose.
El análisis estático del software de control no reveló anomalías.
En definitiva, es razonable admitir que la exigencia está justificada.

Por supuesto, éste es un argumento muy simplificado, y en un argumento de seguridad real deberían presentarse referencias detalladas de la evidencia. Debido a que requieren detalles, los argumentos de seguridad son, por lo tanto, documentos muy extensos y complejos. Están disponibles distintas herramientas software para ayudar a su construcción, y se han incluido enlaces a estas herramientas en las páginas web del libro.

PUNTOS CLAVE

- **Las pruebas estadísticas se utilizan para estimar la fiabilidad del software. Se encargan de probar el sistema con datos de prueba que reflejen el perfil operacional del software. Los datos de prueba pueden generarse automáticamente.**
- **Los modelos de crecimiento de fiabilidad muestran el cambio en la fiabilidad a medida que los defectos son eliminados del software durante el proceso de pruebas. Los modelos de fiabilidad pueden utilizarse para predecir cuándo se alcanzarán los requerimientos de fiabilidad.**
- **Las demostraciones de seguridad son una técnica efectiva de garantía de seguridad de los productos. Muestran que una condición identificada como peligrosa nunca puede ocurrir. Normalmente son más fáciles que probar que un programa satisface sus especificaciones.**
- **Es importante tener un proceso certificado bien definido para el desarrollo de sistemas de seguridad críticos. El proceso debe incluir la identificación y monitorización de contingencias potenciales.**
- **La validación de la seguridad puede realizarse utilizando análisis basado en la experiencia, análisis basado en herramientas, o «equipos tigre» que simulan ataques al sistema.**
- **Los argumentos de seguridad recogen toda la evidencia que demuestra que un sistema es seguro. Estos argumentos de seguridad se requieren cuando un regulador externo debe certificar el sistema antes de que éste sea utilizado.**

LECTURAS ADICIONALES

«Best practices in code inspection for safety-critical software». Este trabajo práctico presenta una lista de recomendaciones para inspeccionar y revisar software de seguridad crítico. [J. R. de Almeida *et al.*, *IEEE Software*, 20(3), mayo-junio de 2003.]

«Statically scanning Java code: Finding security vulnerabilities». Éste es un buen trabajo sobre cómo evitar vulnerabilidades de seguridad en general. Muestra cómo ocurren estas vulnerabilidades y cómo pueden ser detectadas utilizando un analizador estático. [J. Viega *et al.*, *IEEE Software*, 17(5), septiembre-octubre de 2000.]

Software Reliability Engineering: More Reliable Software, Faster Development and Testing. Éste es probablemente el libro definitivo sobre el uso de perfiles operacionales y modelos de fiabilidad para la evaluación de la fiabilidad. Incluye detalles de experiencias con pruebas estadísticas [J. D. Musa, 1998, McGraw-Hill.]

Safety-critical Computer Systems. Este excelente libro de texto incluye un capítulo particularmente bueno sobre el papel de los métodos formales en el desarrollo de sistemas de seguridad críticos. (N. Storey, 1996, Addison-Wesley.)

Safeware: System Safety and Computers. Este trabajo incluye un buen capítulo sobre validación de sistemas de seguridad críticos con más detalle del que se proporciona aquí sobre el uso de argumentos de seguridad basados en árboles de defectos. (N. Leveson, 1995, Addison-Wesley.)

EJERCICIOS

24.1 Describa cómo procedería para validar la especificación de fiabilidad para un sistema de supermercados que usted especificó en el Ejercicio 9.9. Su respuesta debe incluir una descripción de cualquier herramienta de validación que pudiera utilizarse.

24.2 Explique por qué es prácticamente imposible validar las especificaciones de fiabilidad cuando éstas se expresan en términos de un número muy pequeño de fallos sobre el tiempo de vida total de un sistema.

24.3 Utilizando la literatura como información de referencia, escriba un informe para la gestión (para quien no tenga experiencia en esta área) sobre el uso de los modelos de crecimiento de fiabilidad.

24.4 ¿Es ético para un ingeniero el aceptar que se entregue a un cliente un sistema software con defectos conocidos? ¿Existe alguna diferencia si al cliente se le informa de la existencia de estos defectos con antelación? ¿Podría ser razonable imponer exigencias sobre la fiabilidad del software en tales circunstancias?

24.5 Explique por qué el asegurar la fiabilidad del sistema no es una garantía de un sistema seguro.

24.6 El mecanismo de control de bloqueo de puertas en una utilidad para un almacén de desperdicios nucleares se diseña para ser una funcionalidad segura. Dicho mecanismo asegura que la entrada al almacén sólo se permite cuando los escudos de radiación están activados o cuando el nivel de radiación en una habitación caiga por debajo de algún valor determinado (dangerLevel). Es decir:

(i) Si los escudos de radiación remotamente controlados están activados dentro de la habitación, la puerta puede ser abierta por un operador autorizado.
(ii) Si el nivel de radiación está por debajo de un valor determinado, la puerta puede ser abierta por un operador autorizado.
(iii) Un operador autorizado es identificado por la introducción de un código autorizado de entrada de puertas.

```
1     entryCode = lock.getEntryCode () ;
2     if (entryCode == lock.authorisedCode)
3     {
4         shieldStatus = Shield.getStatus ();
5         radiationLevel = RadSensor.get ();
6         if (radiationLevel < dangerLevel)
7             state = safe;
8         else
9             state = unsafe;
10        if (shieldStatus == Shield.inPlace() )
11            state = safe;
12        if (state == safe)
13            {
14                Door.locked = false ;
15                Door.unlock ();
16            }
17        else
18        {
19            Door.lock ( );
20            Door.locked := true ;
21        }
22    }
```

Figura 24.14 Controlador para el bloqueo de puertas.

El código Java mostrado en la Figura 24.14 controla el mecanismo de bloqueo de puertas. Note que el estado seguro es que la entrada no debería permitirse. Desarrolle un argumento de seguridad que muestre que este código es potencialmente no seguro. Modifique el código para hacerlo seguro.

24.7 Usando la especificación para el cálculo de la dosis suministrada dado en el Capítulo 10 (Figura 10.11), escriba un método Java computeInsulin como el usado en la Figura 24.6. Construya un argumento informal de seguridad de que este código es seguro.

24.8 Sugiera cómo podría validar un sistema de protección de contraseñas para una aplicación que usted ha desarrollado. Explique la función de cualquier herramienta que piense que pueda ser útil.

24.9 ¿Por qué es necesario incluir detalles de los cambios del sistema en un argumento de seguridad del software?

24.10 Enumere cuatro tipos de sistemas que podrían requerir argumentos de seguridad del software del sistema.

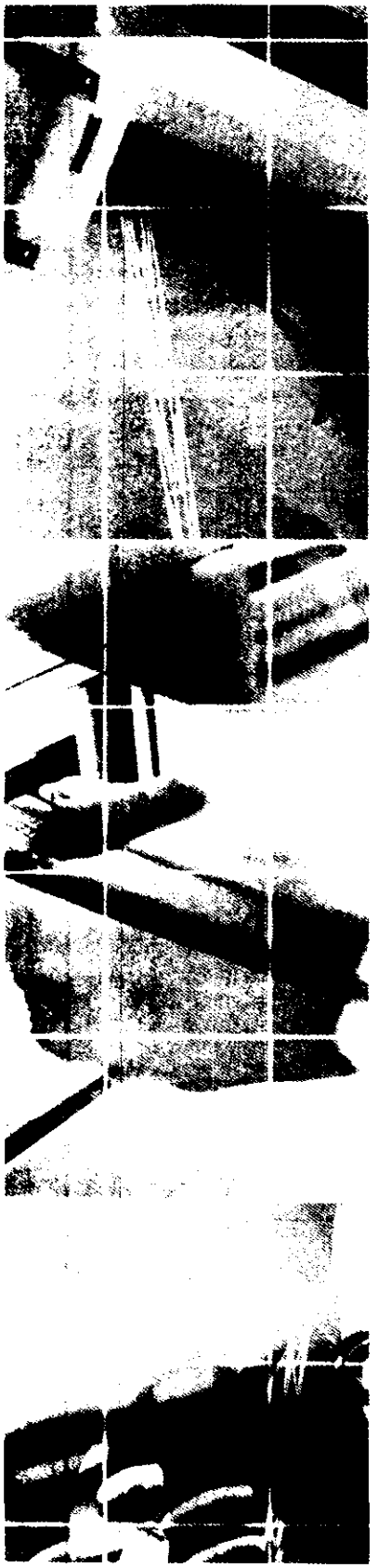
24.11 Suponga que usted formó parte de un equipo que desarrolló software para una planta química, que falló de alguna manera, provocando un serio incidente de contaminación. Su jefe es entrevistado en la televisión y afirma que el proceso de validación ha sido completo y que no existen defectos en el software. Declara que los problemas deben ser debidos a procedimientos de uso no adecuados. Un periodista se acerca a usted para preguntarle su opinión. Comente cómo podría manejar tal entrevista.

PARTE 6 GESTIÓN DE PERSONAL

A veces se sugiere que la diferencia clave entre los ingenieros de software y otros programadores es un proceso gestionado. Esto significa que el desarrollo de software se lleva a cabo dentro de una organización y está sujeta a calendario, presupuesto y restricciones organizacionales. Usted lo puede contrastar con un desarrollo open-source, el cual es una actividad larga y voluntaria. Los desarrolladores de open-source toman sus propias decisiones acerca de cuándo y cómo trabajarán en el desarrollo de este software, y la calidad de su trabajo muestra qué software de calidad no tiene que ser estrictamente gestionado.

Sin embargo, la mayoría de los proyectos no pueden desarrollarse utilizando una aproximación open-source —este modelo de desarrollo sólo es adecuado para sistemas de infraestructuras como sistemas operativos, seridores web, compiladores, etc—. El desarrollo de aplicaciones más especializado es siempre un proceso gestionado. Los capítulos de esta parte del libro amplían la introducción a la gestión del Capítulo 5 y expondrán los siguientes temas:

1. El Capítulo 25 trata acerca de la gestión de personal. No es un tema técnico y por ello no suele aparecer en los libros de ingeniería del software. Sin embargo, bajo mi experiencia, gestionar personal es una actividad crítica en la gestión de proyectos software. Mi objetivo aquí es introducirle en las cuestiones y problemas de la gestión de personal; hablando acerca de la selección y motivación del personal, la gestión de grupos en proyectos y finalmente acerca del modelo SEI de la Madurez de la Capacidad del Personal.
2. En el Capítulo 26 me centraré en la estimación de costes software. Hablaré acerca de la productividad software y de una variedad de técnicas de estimación de costes. Existe mucha incertidumbre en esta área y muchas personas creen que la mejor forma de resolver esta cuestión es con el modelado algorítmico de costes. Yo expondré aquí el modelo COCOMO II, pero como un modelo global, haciendo una breve introducción de sus características.
3. Los Capítulos 27 y 28 tratan de la gestión de la calidad. El Capítulo 27 nos introducirá en las técnicas para garantizar y mejorar la calidad del software. En el Capítulo 28 se vuelve a discutir sobre la mejora de procesos software —como se pueden cambiar lo procesos software para mejorar los atributos de producto y de proceso.
4. Finalmente, el Capítulo 29 habla sobre la gestión de configuraciones. Este tema es importante en sistemas grandes que son desarrollados por equipos. Sin embargo, la necesidad de la gestión de configuraciones no siempre es obvia para los estudiantes que sólo han actuado como personal de desarrollo de software. Yo introduciré varios aspectos en este tema como la planificación de la configuración, gestión de versiones, construcción del sistema y gestión de cambios.



25 Gestión de personal

Objetivos

El objetivo de este capítulo es mostrar la importancia de las personas en el proceso de ingeniería del software. Cuando termine de leer este capítulo:

- comprenderá algunos de los puntos involucrados en la selección y retención de personal en una organización de desarrollo de software;
- comprenderá los factores que influyen en la motivación individual y sus implicaciones para los gestores de proyectos software;
- comprenderá los elementos clave del trabajo en equipo, principalmente la composición, la cohesión, la comunicación y la organización;
- habrá sido introducido en el Modelo de Madurez de la Capacidad del Personal —un modelo que es un marco de trabajo para resaltar las capacidades de los ingenieros del software en una organización.

Contenidos

El personal que trabaja en una organización software es su principal activo. Representa el capital intelectual, y es misión de los gestores asegurar que la organización obtenga los mejores beneficios posibles al invertir en las personas. En las compañías y economías con éxito, esto se cumple cuando las personas son respetadas por la organización. Dichas personas deberán tener un nivel de responsabilidad y se les deberán asignar premios de acuerdo con sus capacidades.

Por lo tanto, la gestión efectiva se refiere a la gestión del personal en una organización. Los gestores de proyectos tienen que resolver problemas técnicos y no técnicos, utilizando a las personas de su equipo de la forma más efectiva posible. Tienen que motivar a la gente, planificar y organizar su trabajo y asegurar que éste se realice de manera adecuada. La mala gestión del personal es uno de los factores más importantes en el fracaso de los proyectos.

Desgraciadamente, esta mala gestión es demasiado común en la industria del software. Los gestores fallan al tomar nota de las limitaciones individuales e imponer metas inalcanzables a los equipos del proyecto. Equiparan la gestión con reuniones, donde la gente colabora en el proyecto. Pueden aceptar nuevos requerimientos sin el análisis adecuado de lo que esto significa para el equipo del proyecto. A veces, ven su papel como una explotación de su personal, en lugar de trabajar con ellos para que su trabajo pueda contribuir a metas tanto organizacionales como personales.

Desde nuestro punto de vista, hay cuatro factores críticos en la gestión de personal:

1. *Objetividad.* El personal del proyecto debe ser valorado de forma equitativa. Mientras que alguien no espere, que todas las recompensas sean idénticas, la gente sentirá que su contribución en la organización es infravalorada.
2. *Respeto.* Las personas tienen diferentes habilidades y los gestores deben respetar estas diferencias. Todos los miembros del equipo deben dar una oportunidad para hacer una contribución. En algunos casos, por supuesto, usted se encontrará con gente que simplemente no se ajusta a un equipo, y no puede continuar, pero es importante no sacar conclusiones acerca de esto.
3. *Incorporación.* La gente contribuye efectivamente cuando siente que se la escucha y se toma nota de sus propuestas. Es importante desarrollar un entorno de trabajo donde todos los puntos de vista, hasta el de la persona más novata, se tomen en consideración.
4. *Honestidad.* El gestor de proyectos siempre debe de ser honesto con lo que está yendo bien y con lo que está yendo mal en el equipo. Debe ser honesto en lo que se refiere a su nivel de conocimientos técnicos y debe dar la razón al personal con más conocimientos cuando sea necesario. Si no es honesto, se encontrará fuera de lugar en algunas ocasiones, y perderá el respeto del grupo.

El objetivo en este capítulo es introducir al lector en algunos elementos que los gestores de proyectos software pueden encontrar, y proveerle de información que le ayude a entender estos elementos. La gestión, desde mi punto de vista, sólo puede ser aprendida con la experiencia; por lo tanto, el papel del libro es ayudarle a aprender de experiencias previas. Para llegar a ser un buen gestor de personal no será suficiente con la simple lectura de este capítulo. Sin embargo, es de esperar que este material ayude a entender los problemas que los gestores encuentran cuando dirigen equipos de individuos con talento técnico.

25.1 Selección de personal

Una de las tareas más importantes de un gestor de proyectos es seleccionar al personal que trabajará en el proyecto. En casos excepcionales, los administradores de proyectos pueden de-

signar a las personas que mejor se adecuan a un trabajo, independientemente de sus otras responsabilidades o de las consideraciones de presupuesto. Sin embargo, por lo general, los administradores de proyectos no tienen una libre elección de personal. Tienen que utilizar a quien esté disponible en una organización, tienen que encontrar a las personas muy rápidamente y cuentan con un presupuesto limitado. Esta limitación de presupuesto restringe el número de costosos ingenieros con experiencia que pueden trabajar en el proyecto.

La decisión de quién será designado para un proyecto, por lo general, se lleva a cabo utilizando tres tipos de información.

1. La información suministrada por los candidatos acerca de sus conocimientos y experiencia (su CV). Ésta suele ser la información más fiable de la que se dispone para juzgar qué candidatos son adecuados.
2. Información obtenida al entrevistar a los candidatos. Las entrevistas pueden darnos una buena visión de qué candidatos son buenos comunicadores y cuándo éstos tienen buenas habilidades sociales. Sin embargo, las pruebas han mostrado que los entrevistadores son obligados a aceptar o rechazar candidatos haciendo rápidos juicios subjetivos. Por lo tanto, las entrevistas no son métodos fiables para juzgar las capacidades técnicas.
3. Recomendaciones de otras personas que han trabajado con los candidatos. Esto puede ser objetivo cuando se conoce a la persona que hace la recomendación. De otro modo, las recomendaciones pueden no ser ciertas, por lo que deben tener poco peso en la decisión a la hora de formar el equipo.

La Figura 25.1 muestra un pequeño caso de estudio de los elementos que podemos encontrar cuando evaluamos personal. Algunas lecciones típicas que muestra el caso de uso son:

1. Si usted está buscando personas con determinadas habilidades dentro de la compañía, el gestor del proyecto en los cuales trabajan estas personas, puede no querer perderlos. A veces, tendremos que aceptar que personas trabajen a tiempo parcial en nuestro proyecto.
2. Pocos candidatos con determinadas habilidades, como diseño de interfaz de usuario e interfaz hardware. Usted puede no tener un amplio abanico de candidatos para estas tareas, especialmente si la compañía no esta cerca de otras industrias software.
3. Los recién titulados pueden no tener las habilidades que usted necesita, pero son normalmente entusiastas y pueden haber tenido contacto con la tecnología más actual.
4. No intente siempre contratar a la persona con más conocimientos técnicos para un trabajo de desarrollo técnico. En el caso de estudio, pudo ser necesaria la interacción con los usuarios ancianos y Alice decidió que Carol sería más comprensiva con sus problemas.

En la Figura 25.2 se muestra una tabla de los factores que pueden influir en su decisión cuando esté seleccionando personal. Los factores más importantes varían dependiendo del dominio de la aplicación, del tipo de proyecto y de las habilidades de los otros miembros del equipo del proyecto.

El gestor de proyectos puede encontrar problemas para descubrir gente con habilidades y experiencia apropiadas. Puede tener que formar un equipo utilizando ingenieros inexperimentados. Esto puede acarrear problemas porque los miembros del equipo no comprenden el dominio de la aplicación o la tecnología. Sin embargo, en proyectos de larga duración, entender los conceptos y el dominio de la aplicación es más importante que los conocimientos específicos, en particular los lenguajes y métodos de programación. Por lo tanto, a menos que

Caso de estudio 1. Eligiendo a los miembros del equipo

Alice es una gestora de proyectos software que trabaja en una compañía que desarrolla sistemas de alarma. Esta compañía desea entrar en el creciente mercado de tecnología de asistencia para ayudar a ancianos y discapacitados a vivir independientemente. A Alice se le encargó formar un grupo de seis desarrolladores para implementar productos basados en la tecnología de alarmas de la compañía. Su primera misión era seleccionar a los miembros del equipo, los cuales podían ya estar dentro de la compañía o fuera de ella.

Para seleccionar al equipo, Alice revisó en primer lugar las habilidades que se iban a necesitar:

- Experiencia con la tecnología de alarmas existente, y su reutilización.
- Experiencia en el diseño de interfaces, ya que los usuarios no van a tener conocimientos y pueden ser discapacitados; de ahí que necesiten facilidades como fuentes de tamaño variable, etc.
- Idealmente, alguien que tenga experiencia en diseño de sistemas basados en tecnología de ayuda a discapacitados. Por otro lado, alguien con experiencia en interfaces con unidades hardware ya que todos los sistemas que se comenzaban a desarrollar incluían algo de control de hardware.
- Habilidades generales de desarrollo.

La siguiente etapa fue encontrar gente en la compañía con las habilidades necesarias. A pesar de que la compañía se había extendido significativamente y tenía poco personal disponible, lo mejor que Alice pudo negociar fue la ayuda de un experto en alarmas (Fred) durante dos días a la semana. Por lo tanto, decidió poner un anuncio para captar el personal para el nuevo proyecto, donde aparecían los atributos que ella deseaba:

- Experiencia en programación en C. Ella había decidido desarrollar todo el software en C.
- Experiencia en diseño de interfaz de usuario. Un diseñador de interfaz de usuario es esencial, pero podría no ser necesario a tiempo completo.
- Experiencia en interfaces hardware con lenguaje C y utilizando sistemas de desarrollo remoto. Todos los dispositivos utilizados tienen interfaces complejos.
- Personalidad comprensiva ya que tenía que relacionarse y trabajar con personas ancianas las cuales iban a proveer los requerimientos y a probar el sistema.

Alice obtuvo 30 respuestas al anuncio, y de los aspirantes, fue posible identificar candidatos adecuados con conocimientos de interfaz con hardware (Dorothy) y experiencia con interfaces de usuario (Ed). Decidió contratar a dos recién titulados (Brian y Bob)que tenían experiencia en programación en C, pero que tendrían que ser formados en la compañía.

Lo único que quedaba era designar a un programador senior para unirse al equipo de desarrollo y Alice tenía dos opciones. Carol contaba con varios años de experiencia en C y recientemente había tenido una parada en su carrera para tener un niño. Dave poseía una experiencia similar en programación y era un entusiasta de la programación. Gastaba mucho de su tiempo trabajando en proyectos open source y tenía un conocimiento teórico de C y C++.

Después de entrevistar a ambos, Alice decidió ofrecer el trabajo a Carol a pesar de que Dave tenía profundos conocimientos de programación.

Figura 25.1 Selección de personal.

se necesiten conocimientos específicos en un lenguaje de programación para un proyecto de corta duración, es mejor elegir a personal con capacidad para resolver problemas o con dominio de la aplicación. Es relativamente sencillo aprender un nuevo lenguaje, pero mucho más difícil desarrollar el conocimiento conceptual necesario para resolver problemas complejos.

Algunas compañías prueban a los candidatos. Incluyen pruebas de aptitud a la programación y pruebas psicotécnicas donde los candidatos completan una serie de ejercicios en un periodo de tiempo relativamente breve. Las pruebas psicotécnicas están dirigidas a conseguir el perfil psicológico del candidato, indicando sus capacidades y habilidades para ciertos tipos de tareas. Algunos gestores consideran que estas pruebas son inútiles; otros piensan que proveen información que ayuda a la selección del personal. Se duda si las pruebas de aptitud pro-

Experiencia en el dominio de la aplicación	Para desarrollar bien un sistema, los desarrolladores deben entender el dominio de la aplicación. Es esencial que algunos miembros del equipo de desarrollo tengan alguna experiencia en el dominio de la aplicación.
Experiencia en la plataforma	Este factor es significante si la programación a bajo nivel es necesaria. En otro caso, no es un atributo crítico.
Experiencia en el lenguaje de programación	Normalmente esto sólo es importante para proyectos de corta duración donde no existe tiempo suficiente para aprender un nuevo lenguaje. Mientras que aprender el lenguaje propiamente dicho no es difícil, empezar a utilizar las librerías y componentes de forma competente puede llevar varios meses.
Habilidad para resolver problemas	Esto es muy importante para ingenieros de software, los cuales tienen que resolver constantemente problemas técnicos. Sin embargo, es casi imposible de juzgar sin conocer el trabajo del candidato.
Soporte educativo	Esto provee un indicador de los fundamentos básicos que el candidato debe conocer y de la habilidad para aprender. Este factor cada vez es más irrelevante, puesto que los ingenieros obtienen experiencia a través de los proyectos.
Habilidad de comunicación	Esto es importante debido a que el personal del proyecto necesita comunicarse oralmente y por escrito con los otros ingenieros, administradores y clientes.
Adaptabilidad	La adaptabilidad se valora observando las diversas experiencias obtenidas por los candidatos. Éste es un atributo importante puesto que indica una habilidad para aprender.
Actitud	El personal del proyecto debe tener una actitud positiva con respecto a su trabajo y debe estar deseoso de aprender nuevas habilidades. Éste es un atributo importante, pero a menudo muy difícil de valorar.
Personalidad	Éste es un atributo importante pero difícil de valorar. Los candidatos deben ser razonablemente compatibles con los otros miembros del equipo. Ningún tipo de personalidad es más o menos adecuada para la ingeniería de software.

Figura 25.2 Factores que determinan la selección de personal.

veen información útil acerca de la capacidad para resolver problemas. La resolución software de problemas complejos es un proceso iterativo de la larga duración. No parece que las habilidades necesarias para la resolución de problemas complejos sean comparables a las habilidades necesarias para completar pruebas de aptitud simples.

La falta de personal técnico con experiencia puede ser resultado de que, en algunas organizaciones, las personas con habilidades técnicas rápidamente alcanzan una meta en sus carreras. Para progresar, deben tomar responsabilidades administrativas. La promoción de estas personas a un status administrativo significa perder el ejercicio de sus habilidades técnicas. Para evitar esta pérdida, algunas compañías desarrollaron estructuras paralelas de carreras técnicas y administrativas de igual valor. Las personas con experiencia técnica se consideran al mismo nivel que los administradores. Si un ingeniero desarrolla su carrera, se puede especializar en actividades técnicas o de gestión y moverse entre ambas sin la pérdida de su status o salario.

25.2 Motivación

Uno de los papeles de los gestores de proyectos es motivar a las personas que trabajan con ellos. Motivación significa organizar el trabajo y su entorno para estimular a las personas a trabajar de la forma más efectiva posible. Si las personas no son motivadas, no se interesarán

en el trabajo que hacen. Éstas trabajarán lentamente, cometerán más errores y no contribuirán a las metas del equipo ni de la organización.

Maslow (Maslow, 1954) sugiere que la gente sea motivada satisfaciendo sus necesidades y que estas necesidades están organizadas en una serie de niveles, que se muestran en la Figura 25.3. El nivel más bajo de esta jerarquía representa las necesidades fundamentales como comer, dormir, etc., y a continuación, en el siguiente nivel, la necesidad de sentirnos seguros en un entorno. Las necesidades sociales se refieren a la necesidad de sentirse parte de un grupo social. Las necesidades de autoestima se refieren a sentirse respetados por los otros, y las necesidades de autorrealización se refieren a desarrollo personal. Las prioridades humanas son satisfacer las necesidades de los niveles inferiores, como estar hambriento, antes de satisfacer las necesidades, más abstractas, de niveles superiores.

Las personas que trabajan en organizaciones de desarrollo de software no están generalmente hambrientas ni sedientas y por lo general no se sienten físicamente amenazadas por su entorno. Por lo tanto, asegurar la satisfacción de las necesidades sociales, de estima y de autorrealización es más importante desde un punto de vista administrativo:

1. Satisfacer las necesidades sociales significa permitir que la gente tenga tiempo para conocer a sus compañeros de trabajo y proveer lugares para que se conozcan. Esto es relativamente fácil cuando todos los miembros del grupo de desarrollo trabajan en el mismo lugar, pero cada vez es más habitual que los miembros del equipo no estén localizados en el mismo edificio o incluso en la misma ciudad o estado. Pueden trabajar en diferentes organizaciones o desde casa la mayor parte del tiempo. Las comunicaciones electrónicas, como el e-mail y la videoconferencia, pueden ser usadas como ayuda al teletrabajo. Sin embargo, la experiencia con comunicaciones electrónicas demuestra que éstas no satisfacen realmente las necesidades sociales. Si el equipo de trabajo está distribuido, deberían organizarse encuentros periódicos cara a cara, para que la gente haga un intercambio de experiencias directamente con los otros miembros del equipo. A través de esta interacción, la gente llega a formar parte de un grupo social y puede ser motivada por las metas y prioridades del grupo.
2. Para satisfacer las necesidades de estima, se necesita mostrar a la gente que es de gran valor para la organización. El reconocimiento público de los logros es una forma sencilla pero efectiva de hacer esto. Obviamente, las personas también deben sentir que se les paga de acuerdo con el nivel que reflejan sus habilidades y experiencia.
3. Finalmente, para satisfacer las necesidades de autorrealización se necesita dar a las personas responsabilidad de su propio trabajo, asignarles tareas cada vez más exigentes (pero no imposibles) y proveerles un programa de capacitación donde puedan desarrollar sus habilidades.

Figura 25.3 Jerarquía de las necesidades humanas.

En la Figura 25.4 se ilustra un problema de motivación con el que se encuentran los gestores frecuentemente. En este ejemplo, un miembro competente del grupo pierde interés en el trabajo y en el grupo. La calidad de su trabajo decae y empieza a ser inaceptable. Esta situación tiene que ser resuelta con rapidez. Si no se zanja el problema, los otros miembros del grupo llegarán a estar descontentos y a sentir que el reparto del trabajo es injusto.

En este ejemplo, Alice trata de descubrir si las circunstancias personales de Dorothy podrían ser el problema. Los problemas personales normalmente afectan a la motivación, ya que la gente no puede concentrarse en su trabajo. Se debe dar tiempo y ayudar a la gente a resolver estas situaciones, aunque también debe dejarse claro que los empleados tienen unas responsabilidades que cumplir.

De hecho, el problema de motivación de Dorothy es uno de los que surgen frecuentemente cuando el proyecto se desarrolla en una dirección imprevista. La gente que espera hacer un tipo de trabajo puede hacer algo totalmente diferente. Esto comienza a ser un problema cuando las personas quieren desarrollar unas habilidades en alguna dirección qué es diferente a la que toma el proyecto. En esas circunstancias, se puede decidir que miembros deberían dejar el equipo y encontrar oportunidades en otro lugar. En esta situación, Alice decide convencer a Dorothy diciéndole que ampliar su experiencia es un paso positivo en su carrera. Ella da a Dorothy más autonomía de diseño y organiza cursos de ingeniería de software que le proporcionen más oportunidades cuando el proyecto haya finalizado.

El principal problema que presenta el modelo de Maslow con la motivación es que toma exclusivamente el punto de vista del individuo. No tiene en cuenta que de hecho la gente se siente parte de una organización, un grupo profesional, y por lo general de una cultura. No se trata simplemente de satisfacer las necesidades sociales; las personas pueden ser motivadas a través de ayudar al grupo a conseguir las metas comunes.

Ser miembro de un grupo cohesivo es altamente motivador para la mayoría de la gente. A las personas con trabajos que no les hacen sentirse realizadas, frecuentemente les gusta ir al trabajo porque están motivadas por personas que trabajan con ellas y por el trabajo que esas personas hacen. Por lo tanto, es tan bueno pensar sobre la motivación individual, como pen-

Caso de estudio 2: Motivación

El proyecto de tecnología de asistencia de Alice empieza bien. Existen buenas relaciones entre los miembros del equipo, y se desarrollan ideas nuevas y creativas. La compañía decide desarrollar un sistema de mensajes peer-to-peer usando televisiones digitales enlazadas a la red de alarmas. Sin embargo, unos meses más tarde, Alice nota que Dorothy, la experta en diseño hardware, empieza a llegar al trabajo tarde, la calidad de su trabajo se deteriora, y cada vez más parece que no se está comunicando con los otros miembros del equipo.

Alice habla acerca del problema informalmente con otros miembros del equipo para tratar de descubrir si las circunstancias personales de Dorothy han cambiado y si es posible que esto la esté afectando en el trabajo. Ellos no saben nada, y Alice decide hablar con Dorothy para tratar de comprender su problema.

Después de negar inicialmente que hay un problema, Dorothy admite que ha perdido interés en su trabajo. Ella esperaba poder utilizar y desarrollar sus habilidades en interfaces hardware. Sin embargo, la dirección que había tomado el producto le dejaba pocas oportunidades para esto. Básicamente, ella trabajaba como programadora en C con otros miembros del equipo. Mientras admite que el trabajo está cambiando, la preocupa que no está desarrollando sus habilidades en interfaces hardware. Está preocupada, porque encontrar un trabajo donde se trabaje con interfaces hardware será difícil después de este proyecto. Como no quiere disgustarse con el equipo revelando que está pensando en el siguiente proyecto, ha decidido que lo mejor es reducir al mínimo las conversaciones con ellos.

Figura 25.4 Motivación del individuo.

sar en cómo el grupo puede ser motivado para alcanzar las metas de la organización. En la sección siguiente se expondrá la gestión de grupos.

En un estudio psicológico de motivación, Bass y Dunteman (Bass y Dunteman, 1963) clasificaron a los profesionales en tres tipos:

1. *Los orientados a tareas*, quienes están motivados por el trabajo que hacen. En la ingeniería del software, son técnicos que están motivados por el reto intelectual de desarrollo software.
2. *Los orientados a sí mismos*, quienes están principalmente motivados por los éxitos y el reconocimiento personal. Están interesados en el desarrollo de software como medio para hacer cumplir sus propios propósitos.
3. *Los orientados a la interacción*, quienes están motivados por la presencia y acciones de sus compañeros de trabajo. Puesto que el desarrollo de software está cada vez más centrado en el usuario, los individuos orientados a la interacción están cada vez más involucrados en ingeniería de software.

Las personalidades orientadas a la interacción por lo general gustan de trabajar como parte de un grupo, mientras que las orientadas a las tareas y las orientadas a sí mismas a menudo prefieren trabajar solas. Es más probable que las mujeres estén más orientadas a la interacción que los hombres. A menudo, ellas son comunicadoras más efectivas. Se hablará de la mezcla de estas personalidades dentro de grupos en la Sección 25.3.2.

Cada motivación individual está compuesta de elementos de cada clase, pero un tipo de motivación es por lo general dominante en un momento dado. Sin embargo, las personalidades no son estáticas y los individuos pueden cambiar. Por ejemplo, el personal técnico que siente que no se le está remunerando debidamente se puede convertir en orientado a sí mismo y anteponer los intereses personales a las cuestiones técnicas.

25.3 Gestionando grupos

Mucho del software profesional es desarrollado por equipos de proyectos que varían en tamaño desde dos hasta varios cientos de personas. Sin embargo, puesto que es claramente imposible que cada persona en un equipo grande trabaje con todos al mismo tiempo y de forma eficaz en un solo problema, estos grandes equipos por lo general se dividen en varios grupos. Cada grupo es responsable de un subproyecto que desarrolla algún subsistema. Como regla general, los grupos de proyectos de ingeniería de software normalmente no tienen más de ocho o diez miembros. Cuando se utilizan grupos pequeños, los problemas de comunicación se reducen. El grupo completo puede reunirse alrededor de una mesa y llevar a cabo reuniones en una oficina.

Por lo tanto, hacer que un grupo trabaje eficazmente es una tarea crítica de gestión. Obviamente, es importante que el grupo tenga el equilibrio correcto de habilidad y experiencia técnicas y de personalidades. Sin embargo, los grupos exitosos son más que una simple colección de individuos con el equilibrio correcto de habilidades. Un buen grupo tiene un espíritu de equipo de tal forma que las personas involucradas se motivan por el éxito del grupo, así como por sus propias metas personales.

Existen varios factores que influyen en el trabajo en grupo:

1. *La composición del grupo.* ¿Existe el equilibrio correcto de habilidades, experiencia y personalidades en el equipo?

2. *La cohesión del grupo.* ¿Piensa el grupo en sí mismo como un equipo más que como una colección de individuos que trabajan juntos?
3. *La comunicación del grupo.* ¿Se comunican los miembros del grupo de forma efectiva uno con otro?
4. *La organización del grupo.* ¿Está organizado el equipo de tal forma que cada uno se siente valorado y satisfecho con su papel en el grupo?

25.3.1 La composición del grupo

Como se comentó en la Sección 25.2, muchos ingenieros de software están motivados fundamentalmente por su trabajo. Los grupos de desarrollo software, por tanto, están frecuentemente compuestos por personas que tienen sus ideas acerca de cómo deben resolverse los problemas técnicos. Éste es un punto a partir del cual aparecen regularmente problemas: los estándares de interfaz se pasan por alto, los sistemas comienzan a rediseñarse según han sido codificados, embellecimientos innecesarios del programa y otros más. A ser posible, se debe tratar de seleccionar un grupo de personas donde se eviten estos problemas.

Un grupo que tiene personalidades complementarias puede trabajar mejor que un grupo seleccionado exclusivamente por sus habilidades técnicas. La gente motivada por su trabajo es la más fuerte técnicamente. Las personas orientadas a sí mismas serán probablemente las mejores para llevar adelante el trabajo hasta finalizar las tareas. La gente orientada a la interacción facilitará las comunicaciones con el grupo. Es particularmente importante tener gente orientada a la comunicación en un grupo. A ellos les gusta hablar con la gente y pueden detectar tensiones y desacuerdos en etapas tempranas, por lo que tienen un gran impacto sobre el grupo.

En el caso de estudio expuesto en la Figura 25.5, se muestra cómo Alice, la gestora del proyecto, ha tratado de crear un grupo con personalidades complementarias. Este grupo en particular tiene una buena mezcla de personas orientadas a la interacción y a las tareas, pero ya se ha indicado en la Figura 25.4 de cómo la personalidad orientada a sí misma de Dorothy puede causar problemas. El papel de Fred con dedicación parcial como experto del dominio puede ser también un problema aquí. El está muy interesado en mejoras técnicas, y puede no interactuar correctamente con otros miembros del grupo. El hecho de que él no sea siempre parte del equipo, implica que él no se identifica bien con las metas del equipo.

Caso de estudio 3: Composición del grupo

En la creación de un grupo para el desarrollo de un sistema de tecnología de asistencia, Alice es consciente de la importancia de la selección de miembros con personalidades complementarias. Mientras estaba entrevistando a la gente, trató de evaluar si los entrevistados orientados a la tarea, orientados a sí mismos u orientados a la interacción. Sintió que al principio ella era del tipo orientado a sí mismo, y que el proyecto le permitiría ser reconocida como gestor senior y ser ascendida. Por lo tanto, buscó a una o dos personas orientadas a la interacción y quería individuos orientados a la tarea para completar el equipo. La evaluación final a la que llegó fue la siguiente:

Alice – orientada a sí misma
Brian – orientado a la tarea
Bob – orientado a la tarea
Carol – orientada a la interacción
Dorothy – orientada a sí misma
Ed – orientado a la interacción
Fred – orientado a la tarea

Figura 25.5 Composición del grupo.

En algunas ocasiones es imposible elegir un grupo con personalidades complementarias. En este caso de estudio, el gestor de proyecto tiene el control del grupo, por lo que las metas individuales no trascienden a los objetivos organizacionales y del grupo. Este control es fácil de lograr si todos los miembros participan en todas las etapas del proyecto. Las iniciativas individuales son más comunes cuando los miembros del grupo están recibiendo instrucciones, sin ser conscientes de la parte que su tarea juega dentro del proyecto.

Por ejemplo, supongamos que a un ingeniero se le da un diseño de un programa para codificar y observa posibles mejoras de diseño. Si estas mejoras se implementan sin comprender la razón del diseño original, podrían tener implicaciones adversas sobre otras partes del sistema. Si todos los miembros del grupo están inmersos en el diseño desde el principio, entenderán las razones de las decisiones de diseño. Se identificarán con estas decisiones en lugar de oponerse a ellas.

Un papel importante es el de líder del grupo. Éste puede ser el responsable de proveer la dirección técnica y gestión del proyecto. Los líderes del grupo deben conservar el trabajo rutinario del grupo, asegurándose de que la gente está trabajando eficazmente y de la forma indicada por el gestor del proyecto en el planning.

Los líderes son normalmente designados por el gestor general del proyecto. Sin embargo, el líder designado puede no ser un verdadero líder en el grupo, siendo más preocupante cuanto más se aleje del liderazgo técnico. Los miembros del grupo pueden ver a otro miembro como líder. Puede tratarse de un ingeniero más competente técnicamente, o puede ser un motivador mejor que el líder del grupo designado.

A veces, resulta eficaz separar al líder técnico de la administración del proyecto. Las personas que son competentes a nivel técnico no siempre son los mejores administradores. El darle un rol administrativo puede reducir su valía dentro del grupo. Es mejor ayudarlo con un administrador que lo libere de las tareas rutinarias.

Imponer a un líder que el grupo no quiera, normalmente causará tensiones. Los miembros del equipo no respetarán al líder y pueden perder la lealtad al grupo a favor de metas individuales. Éste es un problema concreto de las áreas que cambian rápidamente, como es la ingeniería del software, donde los nuevos miembros pueden estar más actualizados y mejor preparados que los líderes experimentados del grupo. Algunas personas con experiencia pueden sentirse mal por la imposición de un líder joven con nuevas ideas.

25.3.2 Cohesión

En un grupo cohesivo, los miembros piensan que es más importante el grupo que los individuos. Los miembros de un grupo cohesivo son leales al grupo. Se identifican con las metas del grupo y con los otros miembros del grupo. Intentan proteger al grupo, como entidad, de interferencias externas. Esto hace que el grupo sea fuerte y que sea capaz de sobrellevar problemas y situaciones inesperadas. El grupo hará frente a los problemas mediante la ayuda y soporte mutuo.

Las ventajas de un grupo cohesivo son las siguientes:

1. *Puede crearse un grupo que utilice estándar de calidad.* Como este estándar se establece por consenso, probablemente se observará mejor que si fuese externo e impuesto al grupo.
2. *Los miembros de un grupo trabajan juntos.* Las personas del grupo pueden aprender unas de otras. Las inhibiciones causadas por la ignorancia son minimizadas, así como el aprendizaje mutuo los anima.

3. *Se puede practicar la programación sin ego puede ser practicada.* Los programas son reconocidos como propiedad del grupo, no de los individuos.

Programación sin ego (Weinberg, 1971) es un estilo de grupo de trabajo en el que los diseños, programas y otros documentos son considerados como una propiedad común del grupo en lugar de el individuo que los ha escrito. En la cultura de programación sin ego, es más probable que las personas ofrezcan su trabajo para inspección por otros miembros, para aceptar críticas y para trabajar con el grupo a fin de mejorar el programa. Los grupos cohesivos son mejores porque todos los miembros sienten que comparten la responsabilidad del software. La idea de la programación sin ego es fundamental en la programación extrema (Beck, 2000), analizada en el Capítulo 17. En la programación extrema, uno de los principios básicos es la mejora constante del código del sistema, independientemente de quién escribió el código.

Además de la mejora de la calidad de los diseños, programas y documentos, la programación sin ego también mejora las comunicaciones dentro del grupo. Ésta permite discusiones desinhibidas sin considerar status, experiencia o sexo. Los miembros cooperan con otros miembros del equipo a través del desarrollo del proyecto. Esto permite que los miembros del equipo trabajen juntos y se sientan como parte de un equipo.

La cohesión del grupo depende de muchos factores, entre los que se encuentran la cultura organizacional y las personalidades del grupo. Los administradores pueden fomentar la cohesión de varias formas. Pueden organizar eventos sociales para los miembros del grupo y sus familias. Pueden tratar de establecer un sentido de identidad del grupo asignándole un nombre y estableciendo un territorio. Pueden estar involucrados en actividades en grupo como deportes y juegos.

Sin embargo, una de las formas más eficaces de promover la cohesión es asegurar que los miembros del grupo sean tratados como responsables y confiables y se les dé acceso a toda la información. A menudo, los administradores sienten que no tienen que revelar cierta información a todo el grupo. Esto invariablemente crea un clima de desconfianza. El intercambio de información simple es una forma barata y eficiente de hacer que la gente sienta que es parte del grupo.

Podemos ver un ejemplo de esto en el caso de estudio de la Figura 25.6. Alice realiza reuniones informales regulares donde comenta al grupo lo que está haciendo. Ella involucra a la gente en el desarrollo del producto, preguntándoles sobre las nuevas ideas derivadas de las ex-

Caso de estudio 4. El espíritu del equipo

Alice, una gestora de proyectos experimentada, entiende la importancia de crear un grupo cohesivo. Como el desarrollo del producto es nuevo, aprovecha la oportunidad de involucrar a todos los miembros del grupo en la especificación y el diseño, obteniendo opiniones sobre la posible tecnología a aplicar con los miembros mayores de sus familias, y trayendo a miembros de su familia al almuerzo semanal del grupo. El almuerzo del grupo es una oportunidad para que todos los miembros del equipo se reúnan informalmente, hablen de sus problemas y se conozcan unos a otros.

En el almuerzo, Alice dice a los miembros del grupo lo que sabe sobre noticias organizacionales, políticas, estrategias, etc. Cada miembro del equipo entonces resume lo que había estado haciendo y el grupo habla acerca de temas generales, como las nuevas ideas de producto para sus parientes mayores.

Cada pocos meses, Alice organiza un «día de visitantes» para el grupo, donde el equipo dedica dos días a actualizar tecnología. Cada miembro del equipo prepara una actualización sobre una tecnología relevante y la presenta al grupo. Ésta es una reunión fuera del lugar de trabajo, en un buen hotel, y el tiempo está programado para el diálogo y la interacción social.

Figura 25.6 Cohesión del grupo.

periencias en su propia familia. Estas reuniones son un buen camino para fomentar la cohesión; las personas se relajan mientras se ayudan mutuamente a aprender las nuevas tecnologías.

Sin embargo, los grupos fuertes y cohesivos algunas veces padecen dos problemas:

1. *Resistencia irracional al cambio de liderazgo.* Si el líder de un grupo cohesivo se reemplaza por alguien externo al grupo, los miembros del grupo se asocian en contra del nuevo líder. Además, dichos miembros invierten tiempo al resistirse a los cambios propuestos por el nuevo líder del grupo, lo que lleva a una pérdida consecuente de la productividad. Por lo tanto, siempre que sea posible, es mejor designar a los nuevos líderes de dentro de los grupos.
2. *Pensamiento de grupo.* El pensamiento de grupo (Janis, 1972) es el nombre que recibe la situación en que las habilidades importantes de los miembros del grupo se corroen por la lealtad al grupo. La consideración de alternativas se reemplaza por lealtad a las normas y decisiones del grupo. Cualquier propuesta favorecida por la mayoría del grupo se adopta sin consideración de alternativas.

Para evitar el pensamiento de grupo, se pueden organizar sesiones formales donde a los miembros del grupo se les pida que critiquen las decisiones. Se pueden introducir expertos para que revisen las decisiones del grupo. Las personas que argumentan, cuestionan y no respetan el *statu quo* se designarán como miembros del grupo. Actúan como un abogado del diablo, constantemente cuestionan las decisiones del grupo, y así fuerzan a otros miembros del grupo a razonar y evaluar sus actividades.

25.3.3 Las comunicaciones del grupo

Es esencial que exista buena comunicación entre los miembros de un grupo de desarrollo de software. Los miembros del grupo deben intercambiar información sobre el estado de su trabajo, las decisiones de diseño que se han tomado y los cambios necesarios para las nuevas decisiones. Las buenas comunicaciones también fortalecen la cohesión puesto que los miembros del grupo comprenden las motivaciones, fortalezas y debilidades de las personas del grupo.

Algunos factores clave que influyen en la efectividad de las comunicaciones son:

1. *El tamaño del grupo.* Si un grupo incrementa su tamaño, es más difícil asegurar que todos los miembros se comuniquen unos con otros de forma efectiva. El número de vínculos de comunicación en una dirección es $n * (n - 1)$ donde n es el tamaño del grupo. Como se puede ver, con un grupo de siete u ocho miembros, es muy posible que alguna persona raramente se comunique. Las diferencias de status entre los miembros del grupo implican que a menudo las comunicaciones son en una sola dirección. Los miembros de status más alto tienden a dominar las comunicaciones con los miembros de status más bajo, quienes a menudo rehúyen iniciar una conversación o hacer observaciones de crítica.
2. *La estructura del grupo.* Las personas en grupos estructurados informalmente se comunican de forma más efectiva que los grupos con una estructura jerárquica formal. En los grupos jerárquicos, las comunicaciones tienden a fluir hacia arriba y hacia abajo de la jerarquía. Las personas al mismo nivel no hablan unas con las otras. Éste es un problema particular en un proyecto grande con varios grupos de desarrollo. Si las personas que trabajan en diferentes subsistemas sólo se comunican con sus administradores, a menudo esto conduce a retrasos y malentendidos.

3. *La composición del grupo.* Si existen demasiadas personas en el grupo que tienen los mismos tipos de personalidad chocan entre ellas y las comunicaciones se inhiben. Por lo general, la comunicación es mejor en grupos de ambos sexos (Marshall y Heslin, 1975) que en grupos de un solo sexo. Las mujeres tienden a estar más orientadas a la interacción que los hombres y los miembros de un grupo de mujeres facilita y dirige la interacción en el grupo.
4. *El entorno de trabajo físico del grupo.* La organización del lugar de trabajo es un factor importante para facilitar o inhibir las comunicaciones. Esto se analiza en la Sección 25.3.5.

25.3.4 La organización del grupo

Los grupos pequeños de programación, por lo general, se organizan de una forma más informal. El líder del grupo se involucra en el desarrollo de software con los otros miembros del grupo. De esta forma, emerge un líder técnico quien controla de forma efectiva la producción de software. En un grupo informal, el trabajo a realizar se discute por el grupo como un todo y las tareas se asignan de acuerdo con la habilidad y experiencia. Los miembros más experimentados del grupo realizan el diseño de sistemas de alto nivel, pero el diseño de bajo nivel es responsabilidad de los miembros a quienes se les asigna una tarea particular.

Los grupos informales pueden ser muy exitosos, particularmente cuando la mayoría de los miembros del grupo son experimentados y competentes. El grupo funciona como una unidad democrática, que toma decisiones por consenso. Psicológicamente, esto mejora el espíritu del grupo con un incremento en la cohesión y el desempeño. Si un grupo se compone en su mayoría de miembros sin experiencia o incompetentes, informalmente pueden ser un estorbo. No existe una autoridad definida para dirigirse al grupo, lo que provoca una falta de coordinación entre los miembros del grupo y, posiblemente, un desastre para éste.

Beck, en su libro sobre programación extrema (Beck, 2000), describe una variante organizacional interesante de la organización democrática de grupos. En este enfoque, muchas decisiones que por lo general se ven como decisiones administrativas —como las decisiones de calendario— son desarrolladas por los miembros del grupo. Los programadores trabajan en parejas para desarrollar el código y tomar una responsabilidad colectiva para los programas a desarrollar.

Como se explica en el Capítulo 26, la habilidad individual influye de forma importante en la productividad del programador. Los mejores programadores pueden tener una productividad 25 veces mayor que los peores programadores. Por lo tanto, lo mejor será utilizar los mejores programadores y darles todo el apoyo necesario.

Para utilizar de forma más efectiva a los programadores con mucha habilidad, Baker (1972) y otros (Anon, 1974; Brooks, 1975) señalaron que los equipos deben construirse alrededor de un programador en jefe con muchas habilidades. El principio subyacente es que el personal con habilidad y experiencia sea el responsable de todo el desarrollo del software. Ellos no deben estar implicados en cuestiones de rutina, y deben tener un buen apoyo técnico y administrativo en su trabajo. Deben centrarse en el software a desarrollar y no involucrarse en reuniones externas.

Pero la organización del equipo tiene serios problemas porque sobrecarga al jefe de programadores y a su asistente. Otros miembros del equipo, a los cuales no se les ha dado la suficiente responsabilidad, comienzan a desmotivarse debido a que sienten que sus habilidades son infrautilizadas.

A pesar de ello, el principio general de incrementar el equipo de programación con especialistas es una buena idea. Cuando se elija a los miembros del equipo, la elección puede cen-

trarse en gente con habilidades genéricas como comunicaciones y resolución de problemas, y luego se puede incorporar a expertos cuando el proyecto lo requiera. El uso de expertos es necesario, y los desarrolladores inexpertos tienen la oportunidad de aprender y desarrollar su pericia a medida que progresa el proyecto.

25.3.5 Entornos de trabajo

El lugar de trabajo tiene efectos importantes en el desempeño de las personas y en su satisfacción en el trabajo. Los experimentos psicológicos han mostrado que el comportamiento se ve afectado por el tamaño de la habitación, el mobiliario, el equipo, la temperatura, la humedad, la luminosidad y la calidad de la luz, el ruido y el grado de privacidad disponible. El comportamiento del grupo se ve afectado por la organización arquitectónica y los recursos de telecomunicaciones. Las comunicaciones dentro del grupo se ven afectadas por la arquitectura del edificio y la organización del lugar de trabajo.

Existe un costo importante y real para proveer buenas condiciones de trabajo. Cuando las personas no están felices con sus condiciones de trabajo, el movimiento de personal se incrementa. Por lo tanto, tiene mucho más costo el reclutamiento y la capacitación. Los proyectos de software se retrasan debido a la falta de personal cualificado (DeMarco y Lister, 1999).

A menudo, el personal de desarrollo de software trabaja en áreas y oficinas abiertas, algunas veces en cubículos, y sólo los administradores principales tienen oficinas individuales. McCue (1978) llevó a cabo un estudio que muestra que la arquitectura abierta que favorecía a muchas organizaciones no era popular ni productiva. Los factores del entorno más importantes que se identificaron en ese estudio de diseño fueron:

1. *Privacidad.* Los programadores requieren un área donde se puedan concentrar y trabajar sin interrupción.
2. *Repercusión del exterior.* Las personas prefieren trabajar con luz natural y con una vista del entorno exterior.
3. *Personalización.* Los individuos adoptan diferentes prácticas de trabajo y tienen diferentes opiniones sobre la decoración. La habilidad para arreglar el lugar de trabajo que permita llevar a cabo éste y personalizar ese entorno es importante.

En resumen, a las personas les gustan las oficinas individuales que puedan organizar como ellos quieran. Existen menos distracciones e interrupciones que en espacios de trabajo abiertos. En las oficinas abiertas, a las personas se les niega la privacidad y un entorno de trabajo silencioso y se les limita la personalización de su lugar de trabajo. La concentración es difícil y el desempeño se degrada.

Proveer oficinas individuales al personal de ingeniería de software puede tener una diferencia significativa en la productividad. DeMarco y Lister (1985) compararon la productividad de los programadores en varios tipos de lugares de trabajo. Observaron que factores como la privacidad del lugar de trabajo y la habilidad para eliminar las interrupciones tenían un efecto importante. Los programadores que tenían buenas condiciones de trabajo eran dos veces más productivos que los programadores con las mismas habilidades pero que tenían condiciones de trabajo peores.

Los grupos de desarrollo necesitan áreas donde todos los miembros del grupo se puedan reunir como grupo y discutir sus proyectos, tanto formal como informalmente. Las salas de juntas deben tener espacio para acomodar a todo el grupo con privacidad. Los requerimientos de privacidad individuales y los requerimientos de comunicación del grupo son objetivos primordiales. McCue sugirió que agrupar oficinas individuales alrededor de las grandes salas de reunión (véase la Figura 25.7) era la mejor solución.

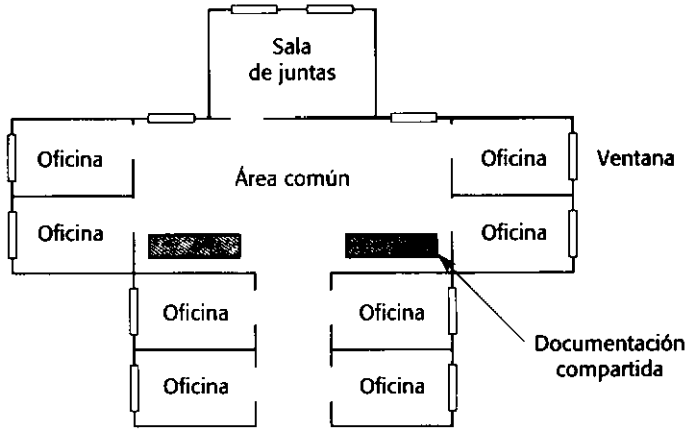


Figura 25.7
Oficina y sala de reuniones del grupo.

Beck (2000) sugirió un modelo similar en su descripción de un entorno para la programación extrema. Sin embargo, sugiere mantener las áreas abiertas, en las que todas las actividades de programación tienen lugar en un área común, y cubículos individuales para los miembros del grupo que deseen trabajar aislados. Claramente, el requerimiento principal es proveer espacios individuales y espacios de grupo para que las personas puedan trabajar solas o en grupo si es necesario.

Este tipo de comunicación ayuda a las personas a resolver sus problemas e intercambiar información de manera informal pero eficaz. Weinberg (Weinberg, 1971) cita un ejemplo anecdótico de cómo una organización quiso hacer que los programadores no desperdiciaran tiempo hablando entre sí alrededor de una máquina de café. Quitaron la máquina, y de forma inmediata tuvieron un notable incremento de solicitudes para asistencia de programación formal. Además de conversar alrededor de la máquina las personas resolvían sus problemas entre sí. Esto ilustra que las compañías necesitan lugares de reunión informales, así como salas de conferencias formales.

El caso de estudio de la Figura 25.8 muestra lo frecuente que es trabajar con restricciones impuestas por los edificios. No es posible adaptarlos o tener todo el espacio que se quisiera. En el ejemplo, Alice usa un despacho individual para trabajos donde se requiere concentración y donde la gente puede discutir en privado qué debe hacer. Los escritorios se comparten

Caso de estudio 5: Organización de la oficina

Alice entiende la importancia de los entornos de trabajo pero su compañía tiene un edificio de los años 70 que no puede ser adaptado a una estructura ideal. Le han asignado tres oficinas para su equipo —una pequeña, individual y separada y dos grandes y juntas, con capacidad cada una para cuatro mesas—. Dos miembros del equipo (Carol y Brian) frecuentemente trabajan desde casa y Fred, el experto en alarmas, sólo trabaja con el equipo dos días por semana. El equipo puede utilizar una sala de reuniones compartida con el resto de los grupos y cada planta del edificio tiene una zona de café para reuniones informales.

En lugar de utilizar la oficina pequeña como su oficina personal, como lo hubiese hecho otro gestor, Alice decide que debe ser una zona tranquila donde el miembro del equipo que lo necesite pueda trabajar sin distracciones. Ella designa una oficina para desarrollo con mesas para el hardware y para los papeles de los prototipos de las interfaces de usuario. Además, esta habitación tiene un escritorio que normalmente usa Fred cuando está trabajando con el equipo, pero también es compartida por Carol y Brian cuando van a trabajar a la oficina. Alice comparte la otra oficina con Bob, Dorothy y Ed. El edificio tiene una red wireless y todos los miembros tienen calculadoras portátiles.

Figura 25.8
Organización de la oficina.

entre los miembros del equipo que no siempre trabajan en la oficina. Cada miembro del equipo tiene una computadora portátil, y puede trabajar en cualquier lugar: en su escritorio, en la habitación tranquila o en zonas comunes del edificio.

25.4 El Modelo de Madurez de la Capacidad del Personal

El Software Engineering Institute (SEI) en Estados Unidos lleva a cabo un programa a largo plazo, de mejora del proceso del software. Parte de este programa es el Modelo de Madurez de la Capacidad (CMM) para el proceso del software que se analiza en el Capítulo 28. Éste se refiere a las mejores prácticas en ingeniería de software. Para soportar este modelo también propusieron un Modelo de Madurez de la Capacidad del Personal (P-CMM) (Curtis *et al.*, 2001). Éste se puede utilizar como un marco de trabajo para mejorar la forma en la que una organización administra sus recursos humanos.

De igual forma que el CMM, el P-CMM es un modelo de cinco niveles como se muestra en la Figura 25.9. Los cinco niveles son:

1. *Inicial Ad hoc*. Se utilizan prácticas informales de administración de personal.
2. *Repetible*. Establece las políticas para el desarrollo de la capacidad del personal.

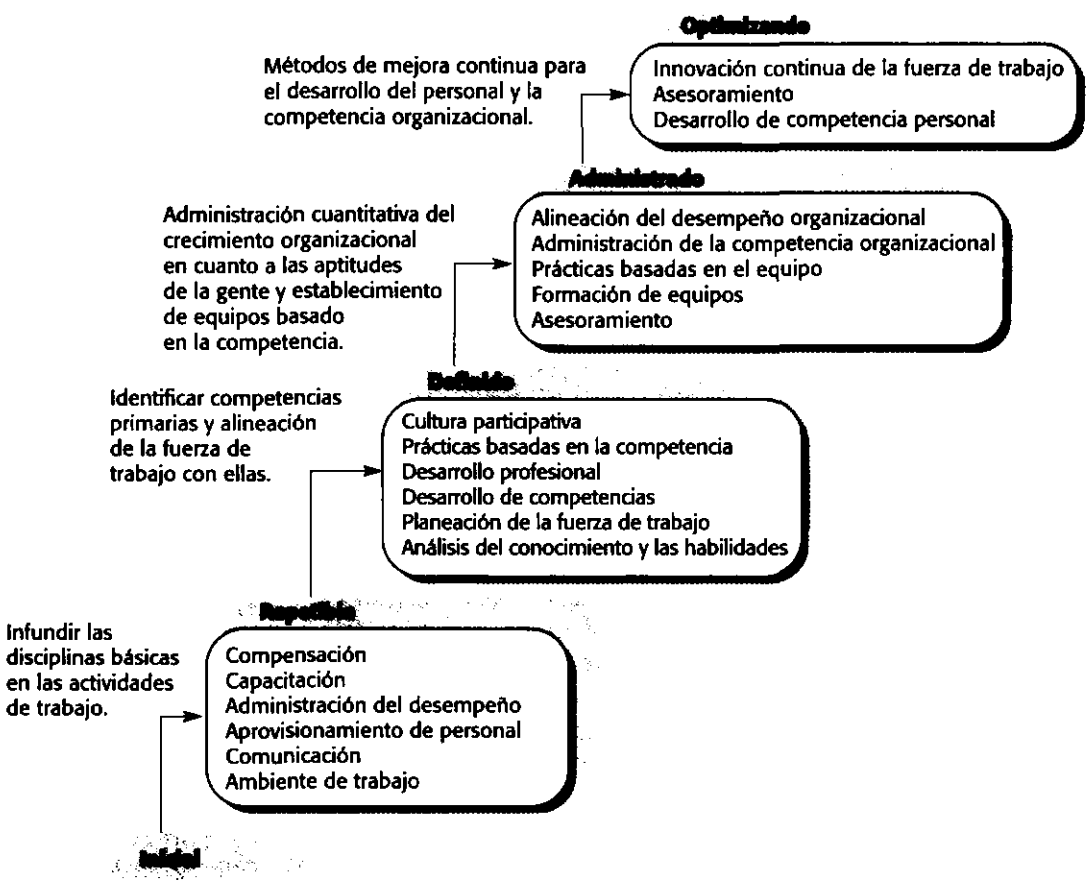


Figura 25.9 El Modelo de Madurez de la Capacidad del Personal.

3. *Definido*. Estandarización de las mejores prácticas de administración de personal a lo largo de la organización.
4. *Administrado*. Se establecen e introducen las metas de la administración del personal.
5. *Optimizado*. Existe un enfoque continuo en la mejora de la competencia individual y la motivación de la fuerza de trabajo.

Curtis *et al.* (Curtis *et al.*, 2001) señalan que los objetivos estratégicos del P-CMM son:

1. Mejorar la capacidad de las organizaciones de software incrementando la capacidad de su fuerza de trabajo.
2. Asegurar que la capacidad de desarrollo de software sea un atributo de las organizaciones más que de unos pocos individuos.
3. Alinear la motivación de los individuos con la de la organización.
4. Retener los activos humanos (por ejemplo, personas con habilidades y conocimiento importantes) dentro de la organización.

El P-CMM es una herramienta práctica para mejorar la administración de las personas en una organización ya que suministra un marco de trabajo para motivar, reconocer, estandarizar y mejorar las buenas prácticas. Refuerza la necesidad de reconocer la importancia de las personas como individuos y de desarrollar sus capacidades. Por supuesto, la aplicación completa de este modelo es muy extensa y probablemente innecesaria para muchas organizaciones. Sin embargo, el modelo es una guía de ayuda que puede conducir a mejoras importantes en la capacidad de las organizaciones para producir software de alta calidad.

PUNTOS CLAVE

- **La selección del personal de un proyecto es una de las tareas más importantes de los gestores de proyectos. Algunos de los factores que pueden usarse para la selección de personal son: experiencia en el dominio de la aplicación, adaptabilidad y personalidad.**
- **La gente es motivada por la interacción con otras personas, por el reconocimiento de los gestores y de sus compañeros, y por las oportunidades de desarrollo personal.**
- **Los grupos de desarrollo software deben ser pequeños y cohesivos. Los líderes del grupo deben ser competentes técnicamente y deben tener ayuda administrativa y técnica.**
- **Las comunicaciones dentro del grupo están influenciadas por factores como el status de los miembros del grupo, el tamaño del grupo, la composición sexual del grupo, las personalidades y los canales de comunicación disponibles.**
- **Los entornos de trabajo para equipos deben incluir espacios donde el equipo pueda interactuar y otros donde los miembros del equipo puedan trabajar individualmente de forma tranquila y concentrándose en su trabajo.**
- **El Modelo de Madurez de la Capacidad del Personal provee un marco de trabajo y sugerencias para mejorar las capacidades de las personas dentro de una organización y mejorar las capacidades de la organización para obtener los máximos beneficios de sus recursos humanos.**

LECTURAS ADICIONALES

A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories. Este libro trata de conclusiones empíricas, hipótesis y teorías relevantes para la ingeniería del software. El Capítulo 10 estudia las habilidades, motivación y satisfacción del usuario y habla de teorías de psicología que refuerza estos temas. (A. Andres y D. Rombach, 2003, Addison-Wesley.)

Software Management, 6th ed. Éste es un tutorial de IEEE que tiene varios artículos sobre cómo dirigir y motivar a las personas. (D. j. Reifer, 2002, Wiley-IEEE Press.)

The People Capability Maturity Model: Guidelines for Improving the Workforce. Este libro hace una descripción fácilmente comprensible del modelo P-CMM, incluyendo una guía para mejorar las capacidades individuales, desarrollar una estructura organizacional fuerte, medir el rendimiento y crear una organización flexible de trabajo. (B. Curtis *et al.*, 2001, Addison-Wesley.)

Peopleware: Productive Projects and Teams, 2nd ed. Éste es un libro clásico sobre la importancia de tratar adecuadamente a las personas cuando gestionamos proyectos de software. Es fácil de leer y es uno de los pocos libros donde se reconoce la importancia del lugar en el que se trabaja. Se recomienda ampliamente. (T. DeMarco y T. Lister, 1999, Dorset House.)

EJERCICIOS

25.1 Explique por qué la coherencia, respeto, inclusión y franqueza son factores que contribuyen eficazmente en la gestión de personal.

25.2 ¿Qué factores deben tenerse en cuenta a la hora de seleccionar el personal para un proyecto software? Razone la respuesta y sugiera cuáles de éstas serán más importantes en la selección de personal para el desarrollo de un sistema empotrado en tiempo real a fin de desarrollar un controlador para un ojo de una máquina de cirugía.

25.3 Desarrolle el caso de estudio del ejemplo sobre motivación de la Figura 25.4 incluyendo actividades generales que Alice podría incorporar para asegurarse de que otros miembros del equipo sigan motivados.

25.4 Explique por qué mantener a los miembros del equipo informados acerca del progreso y de las decisiones técnicas del proyecto puede mejorar la cohesión del grupo.

25.5 Explique lo que entiende por «pensamiento en grupo». Describa los peligros de este fenómeno y explique cómo pueden evitarse.

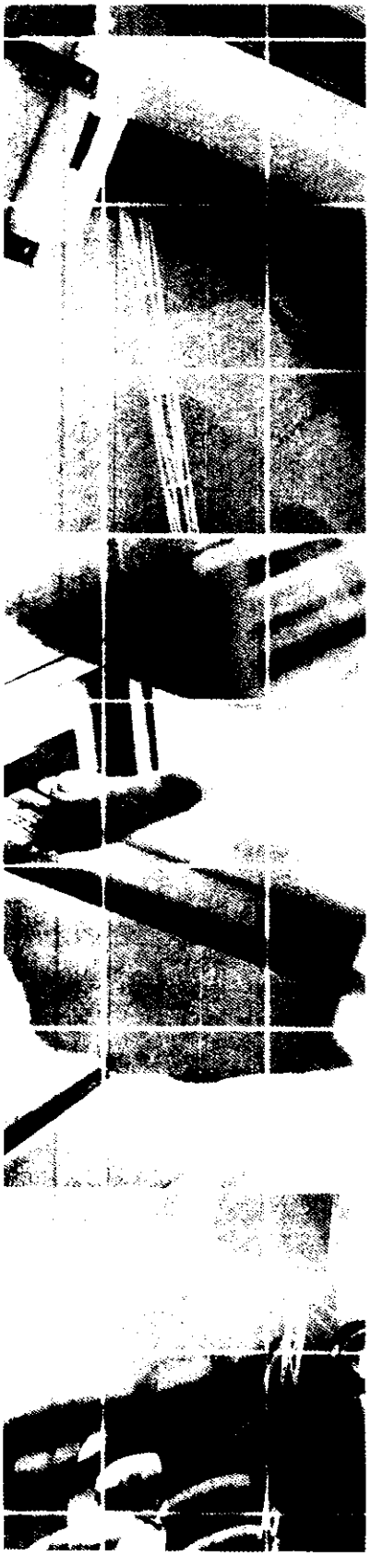
25.6 Explique qué problemas pueden aparecer en un equipo de programación extrema donde muchas decisiones de gestión son tomadas por los propios miembros del equipo.

25.7 ¿Por qué las oficinas de plan abierto y oficinas compartidas son a veces menos recomendables que las individuales para el desarrollo de software? ¿En qué circunstancias piensa que los entornos abiertos pueden ser mejores?

25.8 ¿Por qué es P-CMM un marco eficaz para la mejora de gestión de personal en una organización? Sugiera cómo puede ser modificado para ser usado en pequeñas compañías.

25.9 ¿Deben los gestores hacerse amigos y mezclarse socialmente con los miembros más jóvenes de su equipo de trabajo?

25.10 ¿Es ético suministrar las respuestas que el examinador desea ver más que decir lo que uno siente cuando se llevan a cabo pruebas psicológicas?



26

Estimación de costes del software

Objetivos

El objetivo de este capítulo es introducir las técnicas de estimación de coste y esfuerzo requeridos para el desarrollo de software. Cuando termine de leer este capítulo:

- comprenderá los fundamentos que determinan el coste y esfuerzo del software así como la compleja relación existente entre estos;
- conocerá tres de las métricas que se utilizan para la valoración de la productividad;
- apreciará las diversas técnicas que se utilizan en la estimación de los costes y en la planificación de los proyectos software;
- comprenderá los principios del modelo de COCOMO II para la estimación algorítmica de costes.

Contenidos

En el Capítulo 5, se introdujo el proceso de planificación de proyectos. Durante este proceso, un proyecto se divide en varias actividades que se llevan a cabo en paralelo o de forma secuencial. El análisis previo sobre la planificación de proyectos se centró en la forma de representar las actividades, su dependencia y la asignación de personal para llevar a cabo estas tareas. En este capítulo, se volverá al problema de las estimaciones asociadas al esfuerzo y al tiempo con las actividades identificadas en el proyecto. Cuando estimamos, debemos responder a las siguientes preguntas:

1. ¿Cuánto esfuerzo se requiere para completar una actividad?
2. ¿Cuánto tiempo, de calendario, se necesita para completar una actividad?
3. ¿Cuál es el coste total de una actividad?

La estimación y la creación del calendario del proyecto se llevan a cabo de forma conjunta. Sin embargo, en las primeras etapas del proyecto se requieren algunas estimaciones de costes, antes de que se haga la planificación detallada. Estas estimaciones son necesarias para establecer un presupuesto para el proyecto o para asignar un precio para el software de un cliente.

Existen tres parámetros involucrados en el cálculo del coste total de un proyecto de desarrollo de software.

- Los costes hardware y software, incluyendo el mantenimiento.
- Los costes de viajes y capacitación.
- Los costes de esfuerzo (los costes correspondientes al pago de los ingenieros).

Para muchos proyectos, los costes dominantes son los costes de esfuerzo. Las computadoras con potencia suficiente para desarrollar software son relativamente baratas. Aunque los costes de viajes pueden ser importantes si un proyecto se desarrolla en sitios distintos, son una pequeña parte comparados con los costes de esfuerzo. Además, el uso de correo electrónico, sitios web compartidos y videoconferencias reducen el coste de los viajes y del tiempo hasta en un 50%.

Los costes de esfuerzo no sólo son los salarios de los ingenieros que intervienen en el proyecto. Las organizaciones calculan los costes de esfuerzo en función de los costes totales, donde se tiene en cuenta el coste total para hacer funcionar la organización y dividen éste entre el número de personas productivas. Por lo tanto, los siguientes costes son parte de los costes totales:

1. Costes de proveer, aclimatar e iluminar las oficinas.
2. Los costes del personal de apoyo como administrativos, secretarias, limpiadores y técnicos.
3. Los costes de redes y las comunicaciones.
4. Los costes de los recursos centralizados como las bibliotecas, los recursos recreativos, etc.
5. Los costes de seguridad social, pensiones, seguros privados, etc.

Este factor de sobrecarga normalmente es el doble del salario de un ingeniero software, dependiendo del tamaño de la organización y sus sobrecargas asociadas. Por lo tanto, si a un ingeniero de software se le pagan 90.000 $ al año, el coste total de la organización es de 180.000 $ por año o 15.000 $ por mes.

Una vez que se inicia el proyecto, los gestores deben actualizar regularmente sus estimaciones de tiempo y de coste. Esto ayuda en el proceso de planificación y en el uso efectivo de los recursos. Si los gastos actuales son significativamente mayores que los estimados, el gestor del proyecto deberá tomar alguna decisión. Esto implica que puede incorporar recursos adicionales al proyecto o modificar los trabajos a realizar.

Oportunidad de mercado	Una organización de desarrollo podría ofertar un bajo precio debido a que desea conseguir cuota de mercado. Aceptar un beneficio bajo en un proyecto podría darle la oportunidad de obtener más beneficios posteriormente. La experiencia obtenida le permite desarrollar nuevos productos.
Incertidumbre en la estimación de costes	Si una organización está insegura de su coste estimado, puede incrementar su precio por encima del beneficio normal para cubrir alguna contingencia.
Términos contractuales	Un cliente puede estar dispuesto a permitir que el desarrollador retenga la propiedad del código fuente y que reutilice el código en otros proyectos. Por lo tanto, el precio podría ser menor que si el código fuente del software se le entregara al cliente.
Volatilidad de los requerimientos	Si es probable que los requerimientos cambien, una organización puede reducir los precios para ganar un contrato. Después de que el contrato se le asigne, se cargan precios altos a los cambios en los requerimientos.
Salud financiera	Los desarrolladores en dificultades financieras podrían bajar sus precios para obtener un contrato. Es mejor tener beneficios más bajos de los normales o incluso quebrar antes de quedar fuera de los negocios.

Figura 26.1 Factores que afectan al precio del software.

La estimación del software debe realizarse de forma objetiva e intentando predecir lo mejor posible el coste de desarrollo del software. Si el coste del proyecto se calcula dentro de un presupuesto para un cliente, entonces tendremos que decidir cómo se relacionan ambos conceptos. De forma clásica, el precio es simplemente el coste más el beneficio. Sin embargo, la relación entre el coste del proyecto y su precio no es tan simple.

Al asignar un precio al software debemos tener en cuenta consideraciones organizacionales, económicas, políticas y de negocios. En la Figura 26.1 se muestran los factores que se deben tener en cuenta. Por lo tanto, no existe una relación sencilla entre el precio del software que se le da al cliente y los costes de desarrollo. Debido a las consideraciones organizacionales involucradas, asignar el precio del proyecto por lo general le concierne al administrador principal de la organización, así como a los gestores de software.

Por ejemplo, una pequeña compañía de desarrollo software para empresas de carburantes tiene 10 ingenieros a principios de año, pero para los contratos que tiene actualmente sólo necesita cinco miembros del equipo de desarrollo. Sin embargo, su oferta para un contrato de larga duración con la principal empresa de carburantes requiere 30 personas/año a lo largo de dos años. El proyecto no puede empezar hasta dentro de 12 meses, pero, si se otorga, transformará las finanzas de la pequeña empresa. La compañía software tiene la oportunidad de comenzar un proyecto que requiere seis personas y que debe estar completado en 10 meses. Los costes (incluida la sobrecarga) son estimados en 1,2 millones de dólares. Sin embargo, para mejorar su posición respecto a los competidores, la compañía software ofrece un precio de 0,8 millones. Esto significa que, a pesar de perder dinero en este contrato, la organización puede retener al personal experto para futuros proyectos más rentables.

26.1 Productividad

La productividad en una empresa manufacturera se mide dividiendo el número de unidades producidas entre el número de personas/hora requeridas para producirlas. Sin embargo, para cualquier problema de software, existen muchas soluciones diferentes con distintas caracte-

rísticas. Una solución podría ejecutarse de forma más eficiente mientras otra podría ser más legible y fácil de mantener. Cuando se producen soluciones con distintas características, no tiene sentido comparar las tasas de producción.

Sin embargo, los administradores tienen que estimar la productividad de los ingenieros en el proceso de desarrollo software. Estas estimaciones son necesarias para la estimación del proyecto y para valorar si los procesos o las mejoras tecnológicas son efectivas. Por lo general, estas estimaciones de productividad se basan en medir alguno de los atributos del software y dividir el resultado entre el esfuerzo total requerido para el desarrollo. Existen dos tipos de medidas utilizadas:

1. *Medidas relacionadas con el tamaño.* Ésta se relacionan con el tamaño de la salida de alguna actividad. La medida más común relacionada con el tamaño son las líneas de código fuente entregadas. Otras medidas que se utilizan son el número de instrucciones de código objeto entregado o el número de páginas de la documentación del sistema.
2. *Medidas relacionadas con la función.* Éstas se relacionan con la funcionalidad total del software entregado. La productividad se expresa según la cantidad de funcionalidad útil producida en un tiempo dado. Los puntos de función y los puntos objeto son las medidas más conocidas de este tipo.

Las líneas de código fuente por programador/mes son una métrica ampliamente utilizada en la medida de la productividad. Ésta se calcula contando el número total de líneas de código fuente que se entrega. La cuenta se divide entre el tiempo total de programadores/mes requeridos para completar el proyecto. Por lo tanto, este tiempo incluye el tiempo requerido para el análisis y diseño, codificación, pruebas y documentación.

Este enfoque se desarrolló cuando muchos de los programas estaban en FORTRAN, lenguaje ensamblador o COBOL. Entonces, los programas se tecleaban en tarjetas, con una instrucción en cada tarjeta. El número de líneas de código era fácil de calcular: correspondía al número de tarjetas. Sin embargo, los programas en lenguajes como Java o C++ consisten en declaraciones, instrucciones ejecutables y comentarios. Incluyen macroinstrucciones que ocupan varias líneas de código. Existe más de una instrucción por línea. Por lo tanto, no hay una relación sencilla entre las instrucciones de programa y las líneas de un listado.

Comparar la productividad de los diferentes lenguajes de programación también da impresiones engañosas de la productividad del programador. Cuanto más expresivo sea un lenguaje de programación, más baja será la productividad aparente. Esta anomalía surge debido a que todas las actividades de desarrollo de software se consideran de forma conjunta cuando se calcula la productividad, pero la métrica utilizada sólo tiene en cuenta los procesos de programación. Por lo tanto, si un lenguaje requiere más líneas que otro para implementar la misma funcionalidad, la estimación de la productividad puede ser totalmente errónea.

Por ejemplo, consideremos un sistema que podría codificarse con 5.000 líneas de código ensamblador o 1.500 líneas de código de lenguaje de alto nivel. En la Figura 26.2 se muestra el tiempo de desarrollo para las diversas fases. El programador de ensamblador tiene una pro-

Lenguaje ensamblador	**3 semanas**	**5 semanas**	**8 semanas**	**10 semanas**	**2 semanas**
Lenguaje de alto nivel	**3 semanas**	**5 semanas**	**4 semanas**	**6 semanas**	**2 semanas**

Lenguaje ensamblador	**5.000 líneas**	**28 semanas**	**714 líneas/mes**
Lenguaje de alto nivel	**1.500 líneas**	**20 semanas**	**300 líneas/mes**

Figura 26.2 Tiempos de desarrollo de sistemas.

ductividad de 714 líneas/mes y el programador de alto nivel menos de la mitad de éstas, 300 líneas/mes. Así los costes de desarrollo para un sistema en C son menores y el sistema se desarrolla en menos tiempo.

Una alternativa a la utilización del tamaño del código como atributo de estimación es utilizar alguna medida de la funcionalidad del producto. Esto evitará la anomalía anterior puesto que la funcionalidad es independiente del lenguaje de programación. MacDonell (MacDonell, 1994) describe brevemente y compara varias medidas diferentes basadas en la funcionalidad. La más conocida de estas medidas son los puntos de función. Éstos fueron propuestos por Albrecht (Albrecht, 1979) y refinado por Albrecht y Gaffney (Albrecht y Gaffney, 1983). Garmus y Herron (Garmus y Herron, 2000) describen el uso práctico de los puntos de función en proyectos software.

La productividad se expresa como el número de puntos de función que son implementados por persona/mes. Un punto de función no es una característica simple sino que es una combinación de características del programa. El número total de puntos de función en un programa se calcula midiendo o estimando las siguientes características del programa:

- Entradas y salidas externas.
- Interacciones con el usuario.
- Interfaces externas.
- Archivos utilizados por el sistema.

Obviamente, algunas entradas, salidas e interacciones son más complejas que otras y se tarda más en implementarlas. Las métricas de puntos de función tienen esto en cuenta multiplicando las estimaciones de puntos de función iniciales por un factor de complejidad. Se debería calcular cada una de estas cuestiones y asignarles un valor de peso, que varía desde 3 (para entradas externas simples) hasta 15 (para ficheros internos complejos). Pueden tomarse los pesos propuestos por Albretcht u otros obtenidos mediante la propia experiencia.

Se pueden calcular los llamados puntos de función «no ajustados» (UFC) multiplicando los contadores iniciales por los pesos estimados y sumando todos los valores.

$$\text{UFC} = \sum(\text{número de elementos de un tipo}) \times (\text{peso})$$

Ahora se pueden modificar estos puntos de función «no ajustados» a través de factores de complejidad relativos a la complejidad del sistema como un todo. Esto introducirá en la estimación el grado de proceso distribuido, la cantidad de reutilización y el rendimiento, entre otras cosas. Los puntos de función «no ajustados» se multiplican por estos factores de complejidad del proyecto y producen los puntos de función finales para todo el sistema.

Symons (Symons, 1988) observó que la naturaleza subjetiva de la complejidad de las estimaciones implicaba que la contabilización de puntos de función en un programa depende de la persona que hace la estimación. Diferentes personas tienen nociones diferentes de complejidad. Existe una amplia variedad de contabilización de puntos de función, dependiendo del estimador. Por esta razón, existen versiones que difieren acerca del valor de los puntos de función dependiendo del juicio del estimador y del tipo de sistema a desarrollar. Además, los puntos de función están predispuestos para ser utilizados en sistemas de proceso de datos donde dominan las operaciones de entrada y salida. Es más duro estimar a través de puntos de función sistemas dirigidos por eventos. Por esta razón, algunas personas piensan que los puntos de función no son muy útiles para medir la productividad software (Furey y Kitchenham, 1997; Armour, 2002). Sin embargo, los usuarios de los puntos de función argumentan que, a pesar de sus defectos, éstos son efectivos en situaciones prácticas (Banker *et al.*,1993; Garmus y Herron, 2000).

Los puntos objeto (Banker *et al.*, 1994) son una alternativa a los puntos de función. Éstos pueden ser utilizados en lenguajes 4GL y en lenguajes basados en script. Los puntos objeto no son las clases que pueden ser producidas cuando utilizamos un desarrollo orientado a objetos. El número de puntos objeto en un programa es una estimación de:

1. El número de pantallas independientes que se despliegan. Las pantallas sencillas cuentan como 1 punto objeto, las pantallas moderadamente complejas cuentan como 2 y las pantallas muy complejas cuentan como 3 puntos objeto.
2. El número de informes que se producen. Los informes simples cuentan como 2 puntos objeto, los informes moderadamente complejos cuentan como 5, y los informes que son difíciles de producir cuentan como 8 puntos objeto.
3. El número de módulos en lenguajes imperativos como Java o C++ que deben ser desarrollados para complementar el código de programación de la base de datos se contabilizará como 10 puntos objeto.

Los puntos objeto se utilizan en el modelo de estimación COCOMO II (donde se denominan puntos de aplicación), que se anlizará más adelante en este capítulo. La ventaja de los puntos de objeto sobre los puntos de función es que pueden ser fácilmente estimados a partir de la especificación de alto nivel. Para el cálculo de los puntos objeto sólo intervienen las pantallas, los informes y los módulos escritos en lenguajes de programación convencionales. No se tienen en cuenta detalles de implementación, y la estimación del factor de complejidad es mucho más sencilla.

Si se utilizan los puntos de función o los puntos objeto, entonces se pueden estimar en las primeras etapas del proceso de desarrollo. Las estimaciones de estos parámetros se pueden hacer tan pronto como se diseñen las interacciones externas del sistema. En esta etapa, es muy difícil producir una estimación precisa del tamaño del programa en líneas de código fuente.

El cálculo de los puntos de función y puntos objeto puede utilizarse junto con modelos de estimación de líneas de código. El número de puntos de función se emplea para estimar el tamaño final del código. Utilizando el análisis de datos históricos, se puede estimar el promedio de líneas de código, AVC, para implementar un punto de función en un lenguaje particular. Los valores de AVC varían desde 200-300 LOC/FP en lenguaje ensamblador hasta 2-40 en un lenguaje de programación de base de datos como SQL. El tamaño estimado del código para una nueva aplicación se calcula como se muestra a continuación:

Tamaño del código = AVC × Número de puntos de función

La productividad de los ingenieros que trabajan en una organización se ve afectada por varios factores. Algunos de los más importantes se resumen en la Figura 26.3. Sin embargo, las diferencias individuales en la habilidad son más importantes que cualquiera de estos factores. En una primera valoración de la productividad, Sackman y otros (Sackman *et al.*, 1968) comprobaron que algunos programadores eran diez veces más productivos que otros. La experiencia demuestra que esto sigue siendo cierto. Los equipos grandes tienen una mezcla de habilidades, por lo que tienen una productividad «promedio». Sin embargo, en los equipos pequeños la productividad total depende en gran medida de las aptitudes y habilidades individuales.

La productividad del desarrollo software varía notablemente a través de distintos dominios de la aplicación y organizaciones. Para sistema empotrados, grandes y complejos, la productividad es muy baja, alrededor de 30 LOC/pm. Para aplicaciones bien entendidas, escritas en un lenguaje como Java, ésta puede llegar a ser de 900 LOC/pm. Cuando se mide en

Experiencia en el dominio de la aplicación	Conocer el dominio de la aplicación es esencial para el desarrollo efectivo del software. Los ingenieros que ya conocen un dominio son probablemente los más productivos.
Calidad del proceso	El proceso de desarrollo utilizado puede tener un efecto importante en la productividad. Esto se expondrá en el Capítulo 28.
Tamaño del proyecto	Cuando una proyecto es más grande, se requiere más tiempo para las comunicaciones del equipo. Se dispone de menos tiempo para el desarrollo, por lo que la productividad individual disminuye.
Apoyo tecnológico	La productividad se puede mejorar si se tiene un buen apoyo tecnológico como el de herramientas CASE, de sistemas de gestión de configuraciones, etc.
Entorno de trabajo	Como se explica en el Capítulo 25, un entorno de trabajo silencioso con áreas de trabajo privado contribuye a mejorar la productividad.

Figura 26.3 Factores que afectan a la productividad de ingeniería de software.

términos de puntos objeto, Boehm y otros (Boehm *et al.*, 1995) señalan que la productividad varía desde 4 puntos objeto por mes hasta 50 por mes, dependiendo del tipo de aplicación, las herramientas de apoyo y la capacidad del programador. El problema de estas medidas que expresan el volumen producido en un periodo de tiempo es que no tienen en cuenta las características no funcionales del software como la fiabilidad y el mantenimiento. Implican que más siempre significa mejor. Beck (Beck, 2000), en su estudio sobre programación extrema, establece un excelente punto acerca de la estimación. Si la aproximación está basada en la simplificación y optimización del código, entonces el recuento de número de líneas de código no es relevante.

Además, estas medidas no tienen en cuenta la posibilidad de reutilizar el software producido, utilizar generadores de código u otras herramientas que nos ayuden en la creación de software. Lo que realmente quieren estimar es el coste de crear un sistema particular donde se han definido la funcionalidad, la calidad, el rendimiento, el mantenimiento, etc. Esto sólo se relaciona de manera indirecta con medidas tangibles como el tamaño del sistema.

El gestor no debe usar las medidas de productividad para juzgar las habilidades de los ingenieros de su equipo. Si lo hace, los ingenieros deben estar comprometidos con la calidad con el fin de ser más «productivos». Éste es el caso en el que los programadores «menos productivos» producen código más fiable que es más fácil de comprender y más barato de mantener. Por lo tanto, debería pensar en las medidas de productividad como una información parcial sobre la productividad del programador. Tendrá que considerar otra información, como la calidad de los programas que se producen.

26.2 Técnicas de estimación

No hay una forma simple de hacer una estimación precisa del esfuerzo requerido para desarrollar un sistema de software. Las estimaciones iniciales se hacen partiendo de la definición de requerimientos de usuario de alto nivel. El software tiene que ejecutarse en computadoras poco familiares y utilizar nuevas tecnologías de desarrollo. Probablemente no se conozcan las personas involucradas en el proyecto y sus habilidades. Todos estos factores significan que en una primera etapa del proyecto es difícil producir una estimación precisa de los costes de desarrollo del sistema.

Más aún, existe una dificultad fundamental para valorar la precisión de los diferentes enfoques de las técnicas de estimación de costes. Por lo general, las estimaciones de costes se cumplen por su propia naturaleza. La estimación se utiliza para definir el presupuesto del proyecto y el producto se ajusta para que las cifras del presupuesto se cumplan. No se conocen informes de experimentos de control sobre costes de proyectos donde los costes estimados no se utilicen para ajustar el experimento. Un experimento de control no debería revelar la estimación del coste al administrador del proyecto. Los costes reales tendrían que compararse con los estimados del proyecto. Sin embargo, realizar un experimento de este tipo sería imposible, debido a los altos costes que conlleva y al número de variables que no pueden ser controladas.

A pesar de esto, las organizaciones necesitan hacer estimaciones de esfuerzo y costes. Para hacerlo, se utilizan una o más técnicas de las descritas en la Figura 26.4 (Boehm, 1981). Todas estas técnicas se basan en la experiencia de los gestores de proyectos, los cuales usan sus conocimientos en proyectos previos para llegar a una estimación de los recursos necesarios en un proyecto. Sin embargo, puede haber diferencias importantes entre proyectos pasados y futuros. En los diez últimos años se han introducido muchas técnicas y métodos nuevos de desarrollo. Algunos de los cambios que pueden afectar a las estimaciones basadas en la experiencia son:

1. Sistemas orientados a objetos y distribuidos en lugar de sistemas centralizados (mainframes).
2. Uso de servicios web.
3. Uso de ERP o sistemas basados en bases de datos.
4. Uso de paquetes software ajenos en lugar de desarrollar todo el software propio.
5. Desarrollo reutilizando componentes en lugar de hacer todo un desarrollo nuevo.
6. Desarrollo utilizando lenguajes script tales como TCL o Perl (Ousterhout, 1998).
7. El uso de herramientas CASE o generadores de programas más que desarrollo de software no asistido.

Si los gestores del proyecto no han trabajado con estas técnicas, su experiencia previa no les ayudará a estimar los costes del software. Esto hace que sea más difícil producir costes y estimaciones de agenda precisos.

Los enfoques para la estimación de costes de la Figura 26.4 se pueden abordar utilizando un enfoque descendente o ascendente. Un enfoque descendente se inicia a nivel de sistema. Se comienza examinando la funcionalidad total del producto y su interacción con las subfuncionalidades. Los costes a nivel de sistema tienen en cuenta actividades tales como la integración, configuración, gestión y documentación.

El enfoque ascendente, por el contrario, se inicia a nivel de componentes. El sistema se divide en componentes y se calcula el esfuerzo requerido para desarrollar cada uno de los componentes. Estos costes se suman para dar el esfuerzo requerido del desarrollo del sistema completo.

Las desventajas del enfoque descendente son las ventajas del ascendente, y viceversa. La estimación descendente subestima los costes de resolver problemas técnicos difíciles asociados a componentes específicos como las interfaces de hardware no estándares. No existe justificación detallada de la estimación que se produce. En cambio, la estimación ascendente produce tal justificación y considera cada componente. Sin embargo, este enfoque tiende a subestimar los costes de las actividades de sistema tales como la integración. La estimación ascendente también es más costosa. Debe de haber un diseño inicial del sistema para identificar los componentes a evaluar.

Modelado algorítmico de costes	Se desarrolla un modelo utilizando información histórica de costes que relaciona alguna métrica de software (por lo general, su tamaño) con el coste del proyecto. Se hace una estimación de esa métrica y el modelo predice el esfuerzo requerido.
Juicio experto	Se consultan varios expertos en las técnicas de desarrollo de software propuestas y en el dominio de la aplicación. Cada uno de ellos estima el coste del proyecto. Estas estimaciones se comparan y se discuten. El proceso de estimación se itera hasta que se llega a un consenso.
Estimación por analogía	Esta técnica es aplicable cuando otros proyectos en el mismo dominio de aplicación se han completado. Se estima el coste de un nuevo proyecto por analogía con estos proyectos completados. Myers (Myers, 1989) da una descripción muy clara de este enfoque.
Ley de Parkinson	La Ley de Parkinson establece que el trabajo se extiende para llenar el tiempo disponible. El coste se determinará por los recursos disponibles más que por los objetivos logrados. Si el software se tiene que entregar en 12 meses y se dispone de cinco personas, el esfuerzo requerido se estima en 60 personas/mes.
Pricing to win	El coste del software se estima a partir de a lo que el cliente está dispuesto a pagar por el proyecto. El esfuerzo estimado depende del presupuesto del cliente y no de la funcionalidad del software.

Figura 26.4 Técnicas de estimación de costes.

Cada técnica de estimación tiene sus ventajas e inconvenientes. Cada una utiliza diferente información acerca del proyecto y del equipo de desarrollo; por lo tanto, si se utiliza únicamente un modelo y la información no es precisa, la estimación final será errónea. Para proyectos grandes, se deben utilizar varias técnicas de estimación de costes y comparar sus resultados. Si éstos predicen costes totalmente diferentes, probablemente no tenga información suficiente acerca del producto o del proceso de desarrollo. Se debe buscar más información del producto, del proceso y del equipo y repetir el proceso de cálculo de costes hasta que la estimación converja.

Estas técnicas de estimación son aplicables cuando existe un documento de especificación de requerimientos para el sistema. Éste debe definir todos los requerimientos de usuario y de sistema. Se pueden hacer estimaciones razonables de la funcionalidad del sistema a desarrollar. En general, los proyectos grandes de ingeniería de sistemas tendrán tal documento de requerimientos.

Sin embargo, en muchos casos, los costes de muchos proyectos deberán ser estimados utilizando solamente requerimientos de usuario incompletos. Esto significa que los estimadores tienen muy poca información con la que trabajar. El análisis y la especificación de requerimientos son costosos, y los gestores de la compañía necesitan una estimación inicial del coste del proyecto antes de que puedan tener un presupuesto aprobado para desarrollar requerimientos más detallados o un prototipo del sistema.

En estas circunstancias, «pricing to win» es una estrategia utilizada comúnmente. La noción de «asignar precios para ganar» parece ser inmoral y poco apropiada para los negocios. Sin embargo, tiene algunas ventajas. El coste del proyecto se acuerda en base a un borrador de la propuesta. Entonces se llevan a cabo negociaciones con el cliente para determinar una especificación detallada del proyecto. Esta especificación viene delimitada por el coste acordado. El comprador y el vendedor deben acordar la funcionalidad aceptable del sistema. El factor clave en muchos proyectos no son los requerimientos del proyecto, sino el coste. Los requerimientos pueden cambiarse sin que se exceda el coste.

Por ejemplo, a una compañía se le ofrece un contrato para desarrollar un sistema de entrega de combustible para una compañía de carburantes que periódicamente entrega combusti-

ble a sus estaciones de servicio. No hay un documento detallado de requerimientos para este sistema, por lo que los desarrolladores estiman que un precio de 900.000 $ será suficientemente competitivo y encajará en el presupuesto de la compañía de carburantes. Después de conseguir el contrato, ellos negociarán los requerimientos detallados del sistema y qué funcionalidades básicas serán entregadas; entonces estimarán costes adicionales de otros requerimientos. La compañía de carburantes no perderá necesariamente aquí, porque ha concedido el contrato a una compañía en la que puede confiar. Los requerimientos adicionales pueden ser parte de un futuro presupuesto, por lo que los presupuestos iniciales de la compañía de carburantes no serán perturbados con altos costes iniciales de software.

26.3 Modelado algorítmico de costes

El modelado algorítmico de costes utiliza una fórmula matemática para predecir los costes del proyecto basándose en estimaciones del tamaño del proyecto, el número de ingenieros software, y otros factores de proceso y producto. Un modelo algorítmico de costes puede construirse analizando los costes y los atributos de proyectos completados y encontrando una fórmula que los englobe y aproxime hacia la experiencia actual.

Los modelos algorítmicos de costes se utilizan principalmente para hacer estimaciones de coste de desarrollos software, pero Boehm (Boehm *et al.*, 2000) describe una serie de otros usos para la estimación algorítmica de costes, que incluyen estimaciones para investigadores en compañías software, estimaciones de estrategias alternativas para ayudar a evaluar los riesgos, y estimaciones para la toma de decisiones sobre reutilización, reorganización o subcontración.

En su fórmula más general, una estimación algorítmica de costes puede ser expresada como:

$$\text{Esfuerzo} = A \times \text{Tamaño}^B \times M$$

A es un factor constante que depende de las prácticas organizacionales locales y del tipo de software que se desarrolla. El **Tamaño** es una valoración del tamaño del código del software o una estimación de la funcionalidad expresada en puntos de función o puntos objeto. El valor del exponente **B** por lo general se encuentra entre 1 y 1,5. **M** es un multiplicador generado al combinar diferentes procesos, atributos del producto y del desarrollo, como la dependencia de requerimientos del software y la experiencia del equipo de desarrollo.

La mayoría de los modelos algorítmicos de estimación tienen un componente exponencial (**B** en la ecuación anterior) asociados con la estimación del tamaño. Esto refleja el hecho de que el coste no se incremente linealmente con el tamaño del proyecto. Si el tamaño del software se incrementa, incurrimos en costes extras debidos a la sobrecarga de la comunicación en grandes equipos, gestión de configuraciones más complejas, mayores dificultades en la integración, entre otras. Por lo tanto, cuanto más grande sea el sistema, mayor será el valor de este exponente.

Desafortunadamente, todos los modelos algorítmicos padecen las mismas dificultades básicas:

1. A menudo es difícil estimar el **Tamaño** en una primera etapa de un proyecto donde solamente está disponible la especificación. Las estimaciones de los puntos de función y de los puntos objeto son más fáciles de realizar que las de tamaño del código, pero frecuentemente pueden ser imprecisas.
2. Las estimaciones de los factores **B** y **M** son subjetivas. Las estimaciones varían de una persona a otra, dependiendo de su conocimiento y experiencia.

El número de líneas de código fuente en un sistema terminado es la métrica básica usada en muchos modelos algorítmicos de costes. La estimación del tamaño puede comprender la estimación por analogía con otros proyectos, la estimación de convertir los puntos de función o puntos objeto al tamaño del código, la de los valores del tamaño de los componentes del sistema, y la utilización de un componente de referencia conocido para estimar el tamaño de los componentes, o simplemente puede ser una cuestión de juicio de ingeniería.

La estimación precisa del tamaño del código es difícil en etapas tempranas del proyecto por que el tamaño del código depende de decisiones de diseño que todavía no se han tomado. Por ejemplo, una aplicación que requiere una gestión de datos compleja puede usar una base de datos comercial o implementar su propio sistema gestor de datos. Si se utiliza la base de datos comercial, el tamaño del código será menor, pero puede ser necesario un esfuerzo adicional debido a limitaciones de rendimiento del producto comercial.

El lenguaje de programación utilizado en el desarrollo del sistema también afecta al número de líneas de código a desarrollar. Un lenguaje como Java podría generar más líneas de código que si se utilizara lenguaje C. Sin embargo, este código extra permite más comprobaciones en tiempo de compilación, por lo que los costes de validación probablemente se reducen. ¿Cómo se tiene esto en cuenta? Más aún, también se tiene que estimar la magnitud de la reutilización en el proceso de desarrollo de software y ajustar la estimación del tamaño para tomar ésta en cuenta.

Si se utilizan los modelos algorítmicos para estimar los costes, se debe crear un rango de estimaciones (la peor, la esperada y la mejor) en lugar de una sola estimación y aplicar la fórmula de costes a todas ellas. Las estimaciones son más precisas cuando se conoce el tipo de software a desarrollar, cuando se ha calibrado el modelo utilizando datos locales, y cuando el lenguaje de programación y el hardware han sido predefinidos.

La precisión de las estimaciones producidas por un modelo algorítmico de costes depende de la información del sistema que esté disponible. Conforme avance el proceso software, más información estará disponible y las estimaciones serán más precisas cada vez. Si la estimación inicial del esfuerzo requerido es de x meses de esfuerzo, el rango puede estar comprendido entre $0{,}25x$ y $4x$ en la primera propuesta. Éste se afinará durante el proceso de desarrollo, como muestra la Figura 26.5. Esta figura, adaptada de un artículo de Boehm (Boehm *et al.*, 1995), refleja la experiencia de un buen número de proyectos software. Por supuesto, justo antes de que se entregue el sistema, se puede hacer una estimación muy precisa.

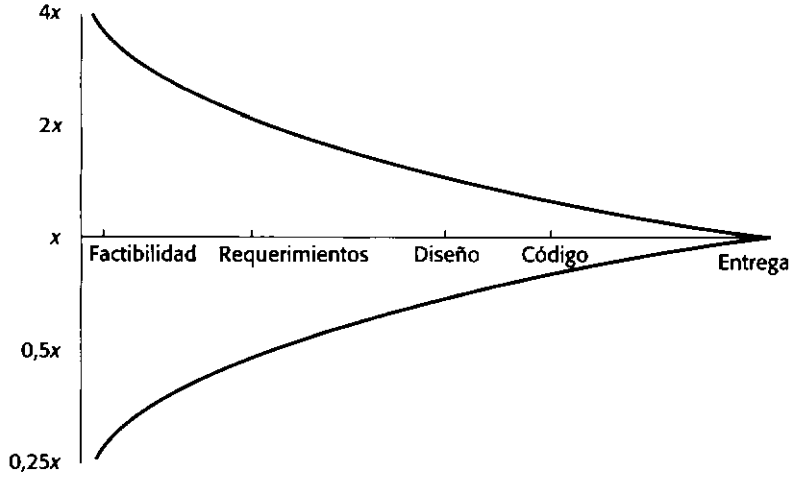


Figura 26.5 Incertidumbre estimada.

26.3.1 El modelo de COCOMO

Se han propuesto varios modelos algorítmicos como base para estimar el esfuerzo, agenda y costes de un proyecto software. Éstos son similares conceptualmente, pero utilizan diferentes valores en sus parámetros. El modelo que se analiza aquí es el modelo COCOMO. El modelo COCOMO es un modelo empírico que se obtuvo recopilando datos de varios proyectos grandes. Estos datos fueron analizados para descubrir las fórmulas que mejor se ajustaban a las observaciones. Estas fórmulas vinculan el tamaño del sistema y del producto, factores del proyecto y del equipo con el esfuerzo necesario para desarrollar el sistema.

Se ha elegido COCOMO por las siguientes razones:

1. Está bien documentado, es de dominio público y lo apoyan el dominio público y las herramientas comerciales.
2. Se ha utilizado y evaluado ampliamente.
3. Tiene una gran tradición desde su primera versión en 1981 (Boehm, 1981), pasando por un refinamiento para el desarrollo de software en ADA (Boehm y Royce, 1989), hasta su versión más reciente, COCOMO II, publicada en 2000 (Boehm *et al.*, 2000).

Los modelos COCOMO son comprensibles, con un gran número de parámetros, los cuales pueden tomar un rango de valores. Éstos son complejos y no se puede dar una descripción completa aquí. Basta una simple exposición de las características esenciales para comprender los modelos algorítmicos de costes.

La primera versión del modelo COCOMO, COCOMO 81, fue un modelo de tres niveles donde éstos reflejaban el detalle del análisis de la estimación del coste. El primer nivel (básico) provee una estimación inicial burda, el segundo nivel la modifica utilizando una serie de multiplicadores de proyecto y proceso, y el nivel más detallado produce estimaciones para las diferentes fases del proyecto. La Figura 26.6 muestra la fórmula básica de COCOMO como los diferentes tipos de proyectos. El multiplicador M refleja características del producto, proyecto y del personal.

COCOMO 81 supone que el software se desarrolla según un proceso en cascada (véase el Capítulo 4) usando lenguajes de programación imperativos estándares como C o FORTRAN. Sin embargo, ha habido cambios radicales en el desarrollo de software desde que se propuso esta versión inicial. El prototipado y el desarrollo incremental son modelos de proceso utilizados comúnmente. El software se desarrolla ahora ensamblando componentes reutilizables y vinculándolos mediante un lenguaje de secuencia de comandos (scripts). Los sistemas que hacen un uso intensivo de datos se desarrollan con lenguajes como SQL y gestores de base de datos comerciales. Se crea nuevo software aplicando reingeniería sobre el software ya existente. Existen herramientas CASE para ayudar en muchas de las actividades del proceso del software.

Simple	**PM = 2,4 (KDSI)^{1,05} × M**	**Aplicaciones bien entendidas y desarrolladas por equipos pequeños.**
Moderado	**PM = 3,0 (KDSI)^{1,12} × M**	**Proyectos más complejos donde los miembros del equipo pueden tener experiencia limitada en este tipo de sistemas.**
Empotrado	**PM = 3,6 (KDSI)^{1,20} × M**	**Proyectos complejos donde el software es parte de un conjunto complejo de hardware, reglas y procedimientos.**

Figura 26.6 Modelo básico de COCOMO 81.

Teniendo en cuenta estos cambios, el modelo COCOMO II considera diferentes enfoques para el desarrollo de software, como el de la construcción de prototipos, el desarrollo basado en componentes y el uso de programación con bases de datos. COCOMO II soporta el modelo de desarrollo en espiral (véase el Capítulo 4) y engloba varios niveles que producen estimaciones detalladas de forma incremental. Éstos pueden utilizarse en sucesivas iteraciones en el desarrollo en espiral. La Figura 26.7 muestra los niveles de COCOMO II y sus recomendaciones de uso.

Los niveles de COCOMO II son los siguientes:

1. *Nivel de construcción de prototipos.* Éste presume que el sistema es creado mediante componentes reutilizables, scripts y programación de base de datos. Fue diseñado para hacer estimaciones de desarrollo de prototipos. Las estimaciones de tamaño del software están basadas en puntos de aplicación, y se utiliza una fórmula simple (tamaño/productividad) para estimar el esfuerzo requerido. El concepto de puntos de aplicación es el mismo que el de puntos objeto expuesto en la Sección 26.1, pero el nombre fue cambiado para evitar confusiones con el desarrollo orientado a objetos.
2. *Nivel de diseño inicial.* Este nivel se utiliza en etapas tempranas del diseño del sistema, después de que los requerimientos hayan sido establecidos. Las estimaciones están basadas en puntos de función, los cuales se convierten a número de líneas de código. La fórmula permite seguir el estándar expuesto anteriormente con un conjunto de siete multiplicadores.
3. *Nivel de reutilización.* Este nivel se utiliza para calcular el esfuerzo requerido para integrar componentes reutilizables y/o el código que es automáticamente generado por herramientas de diseño o programas de traducción. Normalmente es utilizado junto con el nivel de post-arquitectura.
4. *Nivel de postarquitectura.* Una vez diseñado el sistema, se puede hacer una estimación más precisa del tamaño del software. Otra vez se utiliza la fórmula estándar para la es-

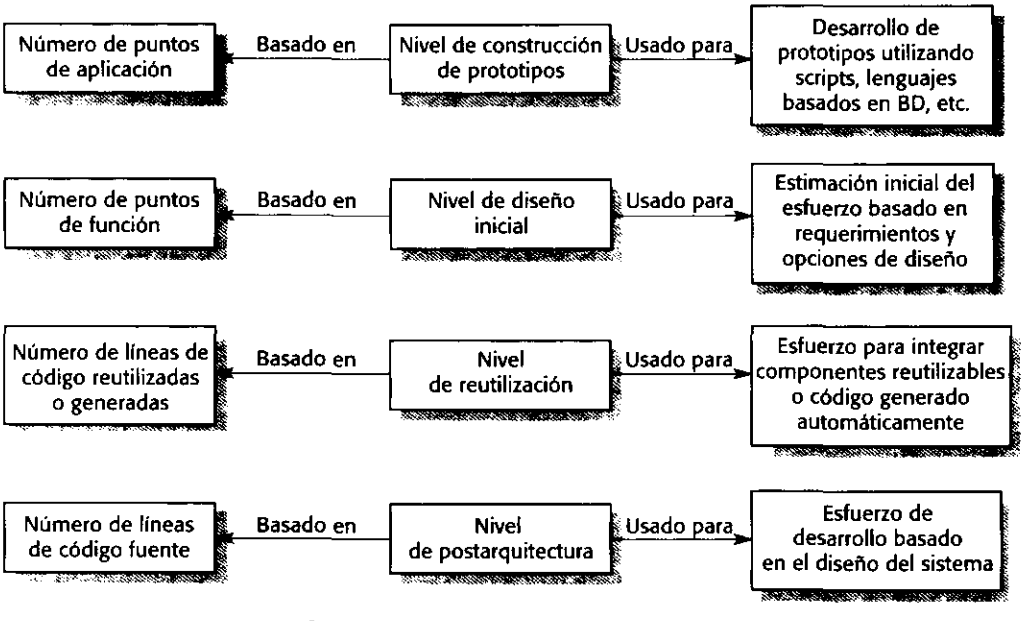


Figura 26.7 Los niveles del COCOMO II.

timación del coste expuesta anteriormente. Sin embargo, ésta incluye un conjunto de 17 multiplicadores que reflejan las habilidades del personal y las características del producto y del proyecto.

Por supuesto, en sistemas grandes, diferentes partes pueden ser desarrolladas utilizando diversas tecnologías, y puede que no sea preciso estimar todas las partes con el mismo nivel de precisión. En estos casos, se puede utilizar el modelo apropiado para cada parte del sistema y combinar los resultados para crear una estimación del sistema completo.

Nivel de construcción de prototipos

El nivel de construcción de prototipos fue introducido en COCOMO II para dar soporte a la estimación del esfuerzo requerido para el prototipado de proyectos y para proyectos en que el software se desarrolla utilizando componentes existentes. Se basa en una estimación de los puntos de aplicación con pesos (puntos objeto), la cual se divide entre una cifra estándar de la productividad estimada. Luego se ajusta la estimación de acuerdo con la dificultad de desarrollo de cada punto objeto (Boehm *et al.*, 2000). La productividad del programador depende de la experiencia y de la capacidad del desarrollador, y de las características de la herramientas CASE utilizadas para apoyar el desarrollo. La Figura 26.8 muestra los valores de productividad sugeridos por los desarrolladores del modelo (Boehm *et al.*, 1995).

En este nivel, la reutilización es común, y algunos de los puntos de aplicación se pueden implementar con componentes reutilizables. En consecuencia, se deberá ajustar la estimación obtenida mediante el total de puntos de aplicación, teniendo en cuenta el porcentaje de reutilización esperado. Por lo tanto, la fórmula para el cálculo del esfuerzo para el prototipado del sistema es:

$$PM = (NAP \times (1 - \%reutilización/100)) / PROD$$

PM es el esfuerzo estimado en personas/mes. **NAP** es el total de puntos de aplicación en el sistema a desarrollar. **%reutilización** es una estimación de la cantidad de código reutilizado en el desarrollo. **PROD** es la productividad medida en puntos objeto como muestra la Figura 26.8. El modelo presupone que no hay esfuerzo adicional derivado de la reutilización.

Nivel de diseño inicial

Este nivel se utiliza cuando hemos acordado los requerimientos de usuario y se han iniciado las primeras etapas del proceso de diseño. Sin embargo, no necesitamos una arquitectura detallada del diseño para realizar estas estimaciones. La meta en este nivel es hacer una estimación aproximada sin demasiado esfuerzo. En consecuencia, se asumirán simplificaciones, como que el esfuerzo de integrar el código reutilizable es cero. Las estimaciones de diseño inicial son la opción más útil para evaluar distintas alternativas a fin de implementar los re-

Experiencia y capacidad de los desarrolladores	Muy baja	Baja	Media	Alta	Muy alta
Madurez y capacidad de las herramientas CASE	Muy baja	Baja	Media	Alta	Muy alta
PROD (NOP/mes)	4	7	13	25	50

Figura 26.8 Productividad medida en puntos objeto.

querimientos del usuario. Las estimaciones en este nivel están basadas en la fórmula estándar para modelos algorítmicos:

$$\text{Esfuerzo} = A \times \text{Tamaño}^B \times M$$

Basándose en su gran cantidad de datos históricos, Boehm propone que el coeficiente **A** sea de 2,94. El tamaño del sistema es expresado en **KSLOC**, es decir, el número de miles de líneas de código fuente. Éste se calcula estimando el número de puntos de función en el software, y convirtiendo éste a **KSLOC** utilizando tablas estándar, que relacionan el tamaño del software con los puntos de función dependiendo del lenguaje de programación.

El exponente **B** refleja el esfuerzo creciente requerido al incrementarse el tamaño del proyecto. Éste no se fija para diferentes tipos de sistemas, como en el COCOMO 81, pero puede variar desde 1,1 hasta 1,24 dependiendo de la novedad del proyecto, la flexibilidad de desarrollo, los procesos utilizados para la resolución de riesgos, la cohesión del equipo de desarrollo y el nivel de madurez del proceso de la organización (véase el Capítulo 28). La forma de calcular este exponente se muestra en la descripción del nivel de postarquitectura del COCOMO II.

El multiplicador **M** en COCOMO II está basado en un conjunto simplificado de siete características de proyecto y de proceso que influyen en la estimación. Éste puede hacer que se incremente o decremente el esfuerzo requerido. Estas características utilizadas en el nivel de diseño inicial son fiabilidad y complejidad del producto (**RCPX**), reutilización requerida (**RUSE**), la dificultad de la plataforma (**PDIF**), la capacidad del personal (**PERS**), la experiencia del personal (**PREX**), agenda (**SCED**) y facilidades de apoyo (**FCIL**). Éstas se pueden estimar directamente sobre una escala de seis puntos donde 1 corresponde a valores muy bajos de los multiplicadores y 6 corresponde a valores muy altos.

Esto nos lleva a la siguiente fórmula de esfuerzo:

$$PM - 2{,}94 \times \text{Tamaño}^B \times M$$

donde:

$$M = \text{PERS} \times \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{PREX} \times \text{FCIL} \times \text{SCED}$$

Nivel de reutilización

Como ya vimos en los Capítulos 18 y 19, ahora es muy común reutilizar software, y los sistemas grandes tienen un porcentaje significativo de código reutilizado de otros proyectos anteriores. Este nivel de reutilización se emplea para estimar el esfuerzo requerido para integrar código reutilizable y código generado.

COCOMO II considera 2 tipos de código reutilizado. El código de caja negra es el código que puede ser reutilizado sin entender el código ni teniendo que hacer cambios en él. El esfuerzo de desarrollo para este tipo de código es 0. El código que ha de ser adaptado para integrarlo con el código nuevo u otros componentes reutilizados recibe el nombre de código de caja blanca. Éste supone un esfuerzo de desarrollo ya que tendremos que entenderlo y modificarlo para que funcione correctamente en el sistema.

Además, muchos sistemas incluyen código generado automáticamente desde el modelo del sistema a través de traductores automáticos. El generador posee unas plantillas estándar, el modelo del sistema es analizado, y el código base es generado a partir de estas plantillas estándar y los datos del modelo. El nivel de reutilización de COCOMO II incluye una parte específica para estimar los costes asociados a este código generado automáticamente.

Para el código generado automáticamente, el modelo estima el número de personas/mes necesarias para integrar este código. La fórmula para estimar el esfuerzo es:

$$PM_{Auto} = (ASLOC \times AT/100) / ATPROD \text{ // Estimación para el código generado}$$

AT es el porcentaje de código adaptado que se genera automáticamente y **ATPROD** es la productividad de los ingenieros que integran dicho código. Boehm *et al.*, 2000) han medido **ATPROD** en torno a 2.400 líneas por mes. Por lo tanto, si hay un total de 20.000 líneas de código reutilizado del tipo caja blanca en un sistema y un 30% de éste se genera automáticamente, entonces el esfuerzo requerido para integrar este código será:

$$(20.000 \times 30/100) / 2400 = 2{,}5 \text{ personas/mes // Ejemplo de código generado}$$

Cuando un sistema incluye algo de código nuevo y algunos componentes reutilizables de caja blanca que deben ser integrados, se usa otro componente del nivel de reutilización. En este caso el nivel de reutilización no calcula el esfuerzo directamente. A pesar de estar basado en el número de líneas de código reutilizadas, éste calcula una cifra equivalente en líneas de código nuevo.

Por lo tanto, si 30.000 líneas de código han de ser reutilizadas, el tamaño nuevo equivalente estimado puede ser 6.000. Esencialmente, reutilizar 30.000 líneas de código equivale a escribir 6.000 líneas de código nuevo. La cifra calculada se suma al número de líneas de código nuevo a desarrollar del nivel de postarquitectura de COCOMO II.

Las estimaciones en el nivel de reutilización son:

ASLOC – Número de líneas de código en los componentes que deben ser adaptados
ESLOC – número equivalente en líneas de código nuevo

La fórmula utilizada para calcular **ESLOC** tiene en cuenta el esfuerzo necesario para comprender el software, para hacer cambios en el código a reutilizar y los cambios en el sistema para integrar este código. También tiene en cuenta la cantidad de código que se genera automáticamente, donde el esfuerzo de desarrollo se calcula como veremos más adelante en esta sección.

La siguiente fórmula se utiliza para calcular el número equivalente de líneas de código:

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

ASLOC se reduce de acuerdo con un porcentaje de código automáticamente generado. **AAM** es el multiplicador de ajuste de la adaptación, el cual tiene en cuenta el esfuerzo requerido en la reutilización de código. Básicamente, **AAM** es la suma de tres componentes:

1. Un componente de adaptación (llamado **AAF**) que representa el coste de hacer los cambios en el código reutilizado. Éste tiene en cuenta cambios de diseño, código e integración.
2. Un componente de comprensión (llamado **SU**) que representa el coste de entender el código que se va a reutilizar y la familiarización del ingeniero con el código. Los valores de **SU** van desde 50 para código complejo no estructurado hasta 10 para código orientado a objetos bien escrito.
3. Un factor de cálculo (llamado **AA**) que representa el coste de la toma de decisiones para la reutilización. Esto es, siempre es necesario algún análisis para decidir que código puede ser reutilizado. **AA** varía entre 0 y 8 según la cantidad de esfuerzo necesaria.

El nivel de reutilización no es un modelo lineal. Necesitaremos esfuerzo si la reutilización se considera como hacer una valoración de si es posible la reutilización. Por lo tanto, cuanta

más reutilización se contemple, los costes de reutilización por unidad de código reutilizado serán mayores.

Nivel de postarquitectura

El nivel de postarquitectura es el nivel más detallado de todos los del COCOMO II. Se utiliza una vez que conocemos el diseño arquitectónico del sistema, es decir, cuando conocemos la estructura de subsistemas.

Las estimaciones producidas en el nivel de postarquitectura están basadas en la misma fórmula básica ($PM = A \times Tamaño^B \times M$) utilizada en las estimaciones de diseño inicial. Sin embargo, la estimación del tamaño del software deberá ser más precisa en esta etapa del proceso de desarrollo, y se utiliza un conjunto más extenso de atributos (17 en lugar de 7) de producto, proceso y organización para refinar el cálculo del esfuerzo inicial. Es posible utilizar más atributos, ya que tenemos más información en esta etapa acerca del proceso de desarrollo y del producto.

La estimación del número de líneas de código se calcula utilizando tres componentes:

1. Una estimación del número total de líneas nuevas de código a desarrollar.
2. Una estimación del número de líneas de código fuente equivalentes (**ESLOC**) calculadas usando el nivel de reutilización.
3. Una estimación del número de líneas de código que tienen que modificarse debido a cambios en los requerimientos.

Estas tres estimaciones se añaden para obtener el tamaño total del código (**KSLOC**) que utilizaremos en la fórmula para el cálculo del esfuerzo. El componente final en la estimación —el número de líneas de código modificado— refleja el hecho de que los requerimientos del software siempre cambian. Los programas tienen que reflejar estos cambios en los requerimientos y, por lo tanto, se ha de desarrollar un nuevo código. Por supuesto, la estimación del número de líneas que cambiarán no es fácil, y habrá más incertidumbre que en las estimaciones de desarrollo.

El término exponencial (**B**) en la fórmula para el cálculo del esfuerzo tiene tres posibles valores en COCOMO I. Éstos se relacionaban con los diferentes niveles de complejidad del proyecto. Puesto que los proyectos son cada vez más complejos, los efectos de incrementar el tamaño del sistema son cada vez más importantes. Sin embargo, las buenas prácticas y los procedimientos organizacionales pueden controlar esta «escala de deseconomía». Esto se recoge en COCOMO II, donde el rango de valores para el exponente B es continuo en lugar de discreto. El exponente se estima considerando 5 factores de escala, como se muestra en la Figura 26.9. Estos factores pueden tomarse de 6 valores que van desde muy bajo hasta extra alto (5 a 0). Los valores resultantes se suman, se dividen entre 100 y al resultado se le suma 1,01 para obtener el exponente que se debe utilizar.

Para ilustrar esto, imaginemos que una organización trabaja en un proyecto en el que se tiene poca experiencia de dominio. El cliente del proyecto no ha definido el proceso a utilizar y no proporciona suficiente tiempo en el calendario del proyecto para hacer un análisis de riesgos. Se tiene que formar un nuevo equipo de desarrollo para implementar este sistema. La organización recientemente ha puesto en proceso un programa de perfeccionamiento y ha obtenido el Nivel 2 del modelo CMM (véase el Capítulo 28). Los posibles valores de los factores de escala utilizados en el cálculo del exponente son:

- *Precedentes:* Éste es un proyecto nuevo para la organización —valor Bajo (4)
- *Flexibilidad de desarrollo:* No se involucra al cliente —valor Muy alto (1)

Precedentes	Refleja la experiencia previa de la organización con este tipo de proyectos. Muy bajo significa sin experiencia previa; Extra alto significa que la organización está completamente familiarizada con este dominio de aplicación.
Flexibilidad de desarrollo	Refleja el grado de flexibilidad en el proceso de desarrollo. Muy bajo significa que se utiliza el proceso de desarrollo descrito; Extra alto significa que el cliente establece sólo metas generales.
Resolución de la arquitectura/riesgo	Refleja la amplitud del análisis de riesgo que se lleva a cabo. Muy bajo significa poco análisis; Extra alto significa un análisis de riesgos completo.
Cohesión del equipo	Refleja cómo de bien se conocen entre sí los miembros del equipo de desarrollo y cómo de bien trabajan juntos; Muy bajo significa interacciones muy difíciles; Extra alto significa un equipo integrado y efectivo sin problemas de comunicación.
Madurez del proceso	Refleja la madurez del proceso de la organización. El cálculo de este valor depende del Cuestionario de madurez del CMM, pero se puede alcanzar una estimación sustrayendo el nivel de madurez del proceso CMM a 5.

Figura 26.9 Factores de escala utilizados en el calculo del exponente en COCOMO II.

- *Resolución de la arquitectura/riesgo:* No se lleva a cabo un análisis de riesgos —valor Muy bajo (5)
- *Cohesión del equipo:* Creación de un equipo, por lo que no tenemos información —valor Nominal (3)
- *Madurez del proceso:* Algún control del proceso en su lugar —valor Nominal (3)

La suma de estos valores es 16, por lo que el exponente debe tomar un valor de 1,17 (0,16 + 1,01).

Los atributos (Figura 26.10) que se utilizan para ajustar las estimaciones iniciales y crear el multiplicador M en el nivel de postarquitectura se puede dividir en cuatro clases:

1. Los atributos de producto se refieren a las características requeridas del producto software a desarrollar.
2. Los atributos de la computadora son restricciones impuestas sobre el software o la plataforma hardware.
3. Los atributos del personal son multiplicadores que tienen en cuenta la experiencia y las capacidades de las personas que trabajan en el proyecto.
4. Los atributos del proyecto se refieren a las características particulares del proyecto de desarrollo software.

La Figura 26.11 muestra un ejemplo de cómo influyen estos conductores de costes en las estimaciones del esfuerzo. Se tomó un valor de 1,17 para el exponente y se supuso que **RELY**, **CPLX**, **STOR**, **TOOL** y **SCED** son los conductores de costes clave en el proyecto. Los otros conductores de costes tienen un valor nominal de 1, por lo que no afectan al cálculo del esfuerzo.

En la citada figura, a los conductores clave de los costes se les asigna valores máximos y mínimos para mostrar cómo influyen en la estimación del esfuerzo. Los valores que se toman son los provenientes del manual de referencia de COCOMO II (Boehm, 1997) Puede verse que los valores altos para los conductores de costes conducen a una estimación del esfuerzo que es tres veces la estimación inicial, mientras que los valores bajos reducen la estimación alrededor de 1/3 de la original. Esto resalta la gran diferencia entre los diferentes tipos de proyectos y las dificultades de transferir la experiencia de un dominio de la aplicación a otro.

RELY	Producto	Fiabilidad requerida del sistema
CPLX	Producto	Complejidad de los módulos del sistema
DOCU	Producto	Amplitud de la documentación requerida
DATA	Producto	Tamaño de la base de datos utilizada
RUSE	Producto	Porcentaje de componentes reutilizables requeridos
TIME	Computadora	Restricciones de tiempo de ejecución
PVOL	Computadora	Volatilidad de la plataforma de desarrollo
STOR	Computadora	Restricciones de memoria
ACAP	Personal	Capacidad de los analistas
PCON	Personal	Continuidad del personal
PCAP	Personal	Capacidad de los programadores
PEXP	Personal	Experiencia de los programadores en el dominio de la aplicación
AEXP	Personal	Experiencia de los analistas en el dominio de la aplicación
LTEX	Personal	Experiencia en el lenguaje y las herramientas de desarrollo
TOOL	Proyecto	Utilización de herramientas de software
SCED	Proyecto	Comprensión de los tiempos de desarrollo
SITE	Proyecto	Ámbito de los distintos lugares de trabajo y sus comunicaciones

Figura 26.10 Conductores del coste del proyecto.

Esta fórmula propuesta por los desarrolladores de COCOMO II refleja su experiencia y datos, pero parece ser muy compleja para su utilización práctica. Existen muchos atributos y demasiadas incertidumbres al estimar sus valores. En principio, cada usuario calibra el modelo y los atributos de acuerdo con sus propios datos históricos de proyecto, y éstos reflejarán las circunstancias particulares dentro del modelo. Sin embargo, en la práctica, pocas organizaciones han recogido suficientes datos de proyectos anteriores de forma que permitan una calibración del modelo. La utilización práctica de COCOMO II empieza normalmente con los valores publicados de los parámetros del modelo, y al usuario le es

Valor del exponente	**1,17**
Tamaño del sistema (incluyendo factores para reutilización y la volatilidad de los requerimientos)	**128.000 DSI**
Estimación inicial de COCOMO sin conductores de coste	**730 personas/mes**
Fiabilidad	Muy alta, multiplicador = 1,39
Complejidad	Muy alta, multiplicador = 1,3
Restricciones de memoria	Alta, multiplicador = 1,21
Utilización de herramientas	Baja, multiplicador = 1,12
Calendario	Acelerada, multiplicador = 1,29
Estimación ajustada de COCOMO	**2306 personas/mes**
Fiabilidad	Muy baja, multiplicador = 0,75
Complejidad	Muy baja, multiplicador = 0,75
Restricciones de memoria	Ninguna, multiplicador = 1
Utilización de herramientas	Muy alta, multiplicador = 0,72
Calendario	Normal, multiplicador =1
Estimación ajustada de COCOMO	**295 personas/mes**

Figura 26.11 El efecto de los conductores de coste sobre las estimaciones de esfuerzo.

imposible saber la similitud de éstos con su situación real. Esto significa que el uso práctico del modelo de COCOMO es limitado. Las grandes organizaciones deben tener los recursos para contratar a un experto en estimación de costes y utilizar los modelos de COCOMO II. Sin embargo, para la mayor parte de las compañías, el coste de calibrar y aprender a usar un modelo algorítmico como el de COCOMO es alto, de forma que no estarán dispuestos a introducir esta aproximación.

26.3.2 Modelos algorítmicos de costes en la planificación

Una de las más valiosas formas de utilizar el modelado algorítmico de costes es comparar las diferentes formas de invertir el dinero para reducir los costes del proyecto. Esto es particularmente importante donde existen costes hardware/software y donde se necesita reclutar personal nuevo con habilidades específicas para el proyecto. El modelo algorítmico ayuda a evaluar los riesgos de cada opción. El coste del modelo nos revela los gastos financieros asociados a las diferentes decisiones de gestión.

Consideremos un sistema empotrado para controlar un experimento que se lanzará al espacio. Los experimentos espaciales tienen que ser muy fiables y están sujetos a límites de peso muy rigurosos. El número de chips en una tarjeta electrónica tiene que minimizarse. En términos del modelo COCOMO los multiplicadores de las restricciones y la fiabilidad son mayores que 1.

Existen tres componentes que hay que tomar en cuenta en el coste de este proyecto:

1. El coste del hardware que ejecuta el sistema.
2. El coste de la plataforma (computadora más hardware) para desarrollar el sistema.
3. El coste del esfuerzo requerido para desarrollar el software.

La Figura 26.13 muestra algunas posibles opciones a considerar. Éstas incluyen gastar más en el hardware para reducir los costes del software o invertir en mejores herramientas de desarrollo.

Los costes de hardware adicionales son aceptables en este caso debido a que el sistema es especializado y no se produce en masa. Sin embargo, si el hardware se incorpora en productos de consumidor, invertir en el hardware para reducir los costes de hardware es raramente aceptable debido a que incrementa el coste unitario del producto.

La Figura 26.13 muestra los costes del hardware, software y totales para las opciones A-F de la Figura 26.12. Aplicar el modelo COCOMO II sin los conductores de costes predice un esfuerzo de 45 personas/mes para desarrollar el software incrustado para esta aplicación. El coste promedio de una persona/mes de esfuerzo es de 15.000 $.

Los multiplicadores relevantes se basan en las restricciones de almacenamiento y tiempo de ejecución (**TIME** y **STOR**), la disponibilidad de herramientas de soporte (compiladores, etcétera) para el desarrollo del sistema (**TOOL**) y la experiencia del equipo de desarrollo en la plataforma (**LTEX**). En todas las opciones, el multiplicador de fiabilidad (**RELY**) es 1,39, que indica que se precisa un esfuerzo adicional significativo para desarrollar un sistema fiable.

El coste del software (**SC**) se calcula como sigue:

$$\text{SC} = \text{Esfuerzo estimado} \times \text{RELY} \times \text{TIME} \times \text{STOR} \times \text{TOOL} \times \text{EXP} \times 15.000\ \$$$

La opción A representa el coste de construcción del sistema con la ayuda y el personal existentes. Representa un punto de comparación. Las demás opciones consideran o más gasto en el hardware o el reclutamiento de nuevo personal (con los costes y riesgos asociados). La opción B muestra que la actualización del hardware no reduce los costes necesariamente. La falta de experiencia del personal incrementa los multiplicadores de experiencia invalidando la

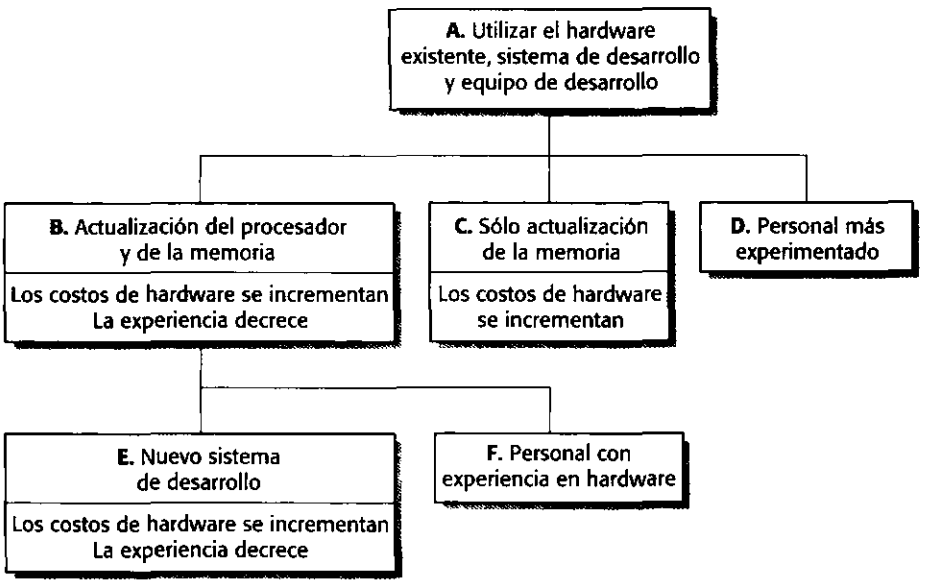


Figura 26.12 Decisiones de gestión.

reducción de los multiplicadores **STOR** y **TIME**. Actualmente es más rentable actualizar la memoria en lugar de actualizar toda la configuración de la computadora.

La opción D parece ofrecer los costes más bajos de todas las estimaciones básicas. No requiere un desembolso adicional de hardware, pero se debe reclutar nuevo personal para el proyecto. Si el personal está disponible en la compañía, ésta es probablemente la mejor opción. Si no es así, se debe reclutar personal del exterior y esto implica costes y riesgos importantes. Esto significa que las ventajas de coste de esta opción son mucho menos significativas que las sugeridas en la Figura 26.13. La opción C ofrece un ahorro de casi 50.000 $ que no implica prácticamente riesgo alguno. Los administradores de proyectos conservadores probablemente seleccionarán esta opción en lugar de la arriesgada opción D.

Las comparativas muestran la importancia de la experiencia del personal como un multiplicador. Si se recluta personal de buena calidad con la experiencia adecuada, se pueden reducir los costes del proyecto de forma significativa. Esto es coherente con el análisis de los factores de productividad de la Sección 26.1. También revela que las inversiones en nuevo hardware y herramientas pueden no ser costeables. Tales estrategias son a menudo promovidas por los desarrolladores a quienes les gusta aprender y trabajar con los nuevos sistemas.

A	1,39	1,06	1,11	0,86	1	63	949.393	100.000	1.049.393
B	1,39	1	1	1,12	1,22	88	1.313.550	120.000	1.402.025
C	1,39	1	1,11	0,86	1	60	895.653	105.000	1.000.653
D	[illegible]								
E	1,39	1	1	0,72	1,22	56	844.425	220.000	1.044.159
F	1,39	1	1	1,12	0,84	57	851.180	120.000	1.002.706

Figura 26.13 Costes de las opciones de gestión.

Sin embargo, la pérdida de experiencia es un efecto más importante en los costes del sistema que los ahorros que provee el uso del nuevo sistema hardware.

26.4 Duración y personal del proyecto

Así como es necesario estimar el esfuerzo requerido para desarrollar un sistema de software y los costes totales del esfuerzo, los gestores de proyectos también estiman cuánto durará el desarrollo del software y cuánto personal se necesita en el proyecto. El tiempo de desarrollo del proyecto se denomina duración del proyecto. Cada vez más, las organizaciones demandan duraciones más cortas para que sus productos salgan al mercado antes que los de sus competidores.

La relación entre el número de personas que trabajan en un proyecto, el esfuerzo total requerido y el tiempo de desarrollo no es lineal. En cuanto crezca el número de personas, se necesitará más esfuerzo. Las personas deben invertir más tiempo en comunicarse y se requiere más tiempo para definir las interfaces entre las partes del sistema. Doblar el número de personas (por ejemplo) no significa que la duración del proyecto se reducirá a la mitad.

El modelo COCOMO incluye una fórmula para estimar el tiempo de calendario (**TDEV**) requerido para completar un proyecto. Esta fórmula es igual para todos los niveles de COCOMO:

$$\mathbf{TDEV} = 3 \times (\mathbf{PM})^{(0{,}33+0{,}2*(B-1{,}01))}$$

PM es el cálculo del esfuerzo y **B** es el exponente calculado (**B** es 1 para el nivel inicial de construcción de prototipos). Este cálculo predice la duración nominal del proyecto.

Sin embargo, la duración prevista del proyecto y la requerida por el plan del proyecto no son necesariamente la misma. La duración planificada es más corta o más larga que la duración media prevista. Sin embargo, existe un límite obvio para los cambios en la duración, y el modelo COCOMO II predice esto como:

$$\mathbf{TDEV} = 3 \times (\mathbf{PM})^{(0{,}33+0{,}2*(B-1{,}01))} \times \mathbf{SCEDPercentage}/100$$

SCDEPercentage es el porcentaje de incremento o decremento en la duración nominal. Si la cifra prevista difiere significativamente de la duración planificada, esto indica que existe un alto riesgo de que surjan problemas para entregar el software como se planeó.

Para ilustrar el cálculo de la duración del desarrollo en COCOMO, supongamos que un proyecto de software requiere 60 meses de esfuerzo de desarrollo (opción C en la Figura 26.12). Supongamos que el valor del exponente B es 1,17. A partir de la ecuación de la duración, el tiempo requerido para completar el proyecto es:

$$\mathbf{TDEV} = 3 \times (60)^{0{,}36} = 13 \text{ meses}$$

En este caso, no existe compresión o expansión de la duración, por lo que el último término de la fórmula no tiene efecto en los cálculos.

Una implicación interesante del modelo COCOMO es que el tiempo requerido para completar el proyecto está en función del esfuerzo total requerido para el proyecto. No depende del número de ingenieros de software que trabajen en el proyecto. Esto confirma la noción de que agregar más personas al proyecto probablemente no ayudará a reducir la duración. Myers (Myers, 1989) analiza los problemas de aceleración de la duración. Señala que en los proyectos de software aparecen problemas importantes si se trata de desarrollarlos sin permitir suficiente tiempo de calendario.

Dividir el esfuerzo requerido en un proyecto por la duración del proyecto no da una indicación útil del número de personas requeridas para el equipo del proyecto. Generalmente, un

pequeño número de personas es necesario para entregar el diseño inicial del proyecto. El equipo crece durante la implementación y las pruebas del sistema, y finalmente vuelve a decrecer. Un crecimiento muy rápido del personal del proyecto a menudo está correlacionado con retrasos en la duración del proyecto. Por lo tanto, los administradores del proyecto deben evitar agregar mucho personal a un proyecto en las etapas iniciales de su ciclo de vida.

El esfuerzo se puede modelar por la llamada curva de Rayleigh (Londeix, 1987) y el modelo de estimación de Putnam (Putnam, 1978), el cual incorpora un modelo para el personal del proyecto que se basa en estas curvas. El modelo de Putnam también incluye el tiempo de desarrollo como factor clave. Si se reduce el tiempo de desarrollo, el esfuerzo requerido para desarrollar el sistema crece exponencialmente.

PUNTOS CLAVE

- **No hay una relación sencilla entre el precio de un sistema y sus costes de desarrollo. Los factores organizacionales pueden hacer que el precio se incremente para compensar los riesgos o se decremente para ser más competitivo.**
- **Los factores que afectan la productividad incluyen la aptitud individual (el factor dominante), la experiencia en el dominio, el proceso de desarrollo, el tamaño del proyecto, la herramienta de apoyo y el entorno de trabajo.**
- **El precio del software es estipulado frecuentemente para conseguir un contrato, y la funcionalidad del sistema es ajustada al precio estimado.**
- **Existen varias técnicas para la estimación de costes de software. Para preparar una estimación, se deben utilizar varias de estas técnicas. Si las estimaciones divergen ampliamente, esto significa que se dispone de información inadecuada para la estimación.**
- **El modelo de costes de COCOMO II es un modelo bien desarrollado que tiene en cuenta el proyecto, el producto, el hardware y los atributos de personal en sus fórmulas de estimación del coste. También incluye un medio para estimar la duración del desarrollo.**
- **Los modelos algorítmicos de costes permiten analizar las opciones cuantitativamente. Permiten calcular el coste de las diferentes opciones y, aunque con errores, las opciones se pueden comparar sobre una base objetiva.**
- **El tiempo requerido para completar un proyecto no es proporcional al número de personas que trabajan en él. Agregar más personal a un proyecto retrasado puede retrasar aún más la fecha de finalización.**

LECTURAS ADICIONALES

«Ten unmyths of project estimation». Un artículo pragmático en el que se habla acerca de las dificultades prácticas de la estimación de costes y cambia algunos supuestos fundamentales en esta área. [P. Armour, *Comm. ACM*, 45(11), noviembre de 2002.]

Software Cost Estimation with COCOMO II. Éste es el libro definitivo sobre el modelo COCOMO II. En él se hace una descripción exhaustiva del modelo con ejemplos e incorpora software que implementa el modelo. Es muy detallado y fácil de leer. (B. Boehm *et al.*, 2000, Prentice Hall.)

Software Project Management: Readings and Cases. Una selección de artículos y casos de estudio sobre la gestión de proyectos de software que es particularmente buena en el tratamiento del modelado algorítmico de costes. [C. F. Kemerer (ed.), 1997, Irwin.]

«Cost models for future software life cycle processes: COCOMO II». Una introducción del modelo de estimación de costes COCOMO II que incluye el razonamiento fundamental de las fórmulas utilizadas. (B. Boehm *et al., Annals of Software Engineering*, 1, Balzer Science Publishers, 1995.)

EJERCICIOS

26.1 En qué circunstancias puede una compañía fijar un precio mucho más alto, para un sistema software, que el sugerido por la estimación de costes más un considerable margen de beneficio.

26.2 Describa dos métricas utilizadas para medir la productividad de los programadores. Comente brevemente las ventajas e inconvenientes de estas métricas.

26.3 Para el desarrollo de sistemas empotrados en tiempo real grandes, sugiera cinco factores que probablemente tengan un efecto significativo en la productividad del equipo de desarrollo de software.

26.4 Las estimaciones de los costes son inherentemente arriesgadas, con independencia de la técnica de estimación utilizada. Sugiera cuatro formas que permitan reducir los riesgos en una estimación del coste.

26.5 ¿Por qué deben utilizarse varias técnicas de estimación en sistemas grandes y complejos?

26.6 Un administrador de software está a cargo del desarrollo de un sistema de software de seguridad crítico que se diseña para controlar una máquina de radioterapia para tratar a los pacientes que sufren de cáncer. Este sistema está incrustado en la máquina y debe ejecutarse en un procesador de propósito especial con una cantidad fija de memoria (8 MB). La máquina se comunica con un sistema de base de datos de pacientes para obtener los detalles del paciente y, después del tratamiento, automáticamente registra la dosis de radiación suministrada y otros detalles del tratamiento en la base de datos.

El método COCOMO se utiliza para estimar el esfuerzo requerido para desarrollar este sistema y se calcula un estimado de 26 personas/mes. Todos los multiplicadores conductores de costes se ajustan a 1 cuando se hace esta estimación.

Explique por qué esta estimación debe ajustarse para tener en cuenta al proyecto, al personal, al producto y a los factores organizacionales. Sugiera cuatro factores que podrían tener efectos importantes en la estimación inicial de COCOMO y proponga valores posibles de estos factores. Justifique por qué se ha incluido cada factor.

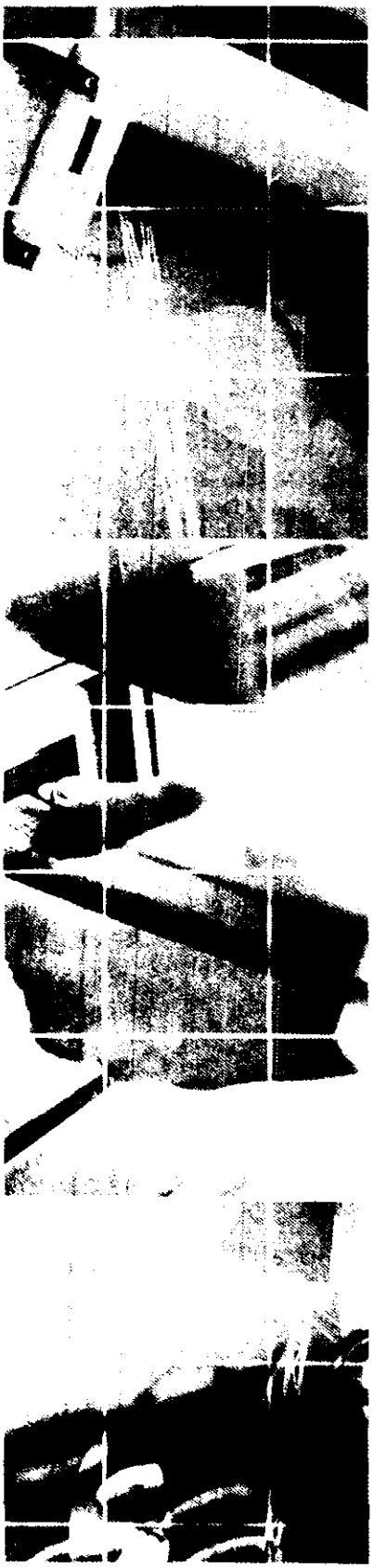
26.7 Dé tres razones por las cuales las estimaciones algorítmicas de costes realizadas por diferentes organizaciones no son directamente comparables.

26.8 Explique cómo utilizan los administradores el enfoque algorítmico para la estimación de costes para el análisis de opciones. Sugiera una situación en la que los administradores elijan un enfoque que no se base en el coste más bajo del proyecto.

26.9 Algunos proyectos de software muy grandes implican escribir millones de líneas de código. Indique en qué medida pueden ser útiles los modelos de estimación de costes para tales sistemas. ¿Por qué las suposiciones en las que se basan podrían no ser válidas para sistemas de software muy grandes?

26.10 ¿Es ético que una empresa dé un precio bajo para un contrato de software sabiendo los requerimientos son ambiguos y que fijarán un alto precio para los cambios subsecuentes requeridos por el cliente?

26.11 ¿Los administradores deberían utilizar la productividad medida durante el proceso de evaluación del personal? ¿Qué salvaguardas son necesarias para asegurar que la calidad no se vea afectada por esto?



27 Gestión de la calidad

Objetivos

El objetivo de este capítulo es introducirnos en la gestión de la calidad y la medición del software. Cuando termine de leer este capítulo:

- entenderá el proceso de gestión de la calidad y las actividades del proceso de garantía, planificación y control de la calidad;
- entenderá la importancia de los estándares en el proceso de gestión de la calidad;
- entenderá qué son las métricas de la calidad y las diferencias entre métricas de predicción y métricas de control;
- entenderá cómo las mediciones pueden ser útiles en el cálculo de los atributos de calidad del software;
- será consciente de las actuales limitaciones de la medición del software.

Contenido

La calidad del software se ha mejorado significativamente en los quince últimos años. Una de las razones ha sido que las compañías han adoptado nuevas técnicas y tecnologías como el uso de desarrollo orientado a objetos y el soporte asociado de herramientas CASE. No obstante, también ha habido una mayor conciencia de la importancia de la gestión de la calidad y de la adopción de técnicas de gestión de la calidad para desarrollo en la industria del software.

Sin embargo, la calidad del software es un concepto complejo que no es directamente comparable con la calidad de la manufactura de productos. En la manufacturación, la noción de calidad viene dada por la similitud entre el producto desarrollado y su especificación (Crosby, 1979). En un mundo ideal, esta definición debería aplicarse a todos los productos, pero, para sistemas de software, existen estos problemas:

1. La especificación se orienta hacia las características del producto que el consumidor quiere. Sin embargo, la organización desarrolladora también tiene requerimientos (como los de mantenimiento) que no se incluyen en la especificación.
2. No se sabe cómo especificar ciertas características de calidad (por ejemplo, mantenimiento) de una forma no ambigua.
3. Como se indicó en la Parte 1, en la que se estudió la ingeniería de requerimientos, es muy difícil redactar especificaciones concretas de software. Por lo tanto, aunque un producto se ajuste a su especificación, los usuarios no lo consideran un producto de alta calidad debido a que no responde a sus expectativas.

Se deben reconocer estos problemas con la especificación del software y se tienen que diseñar procedimientos de calidad que no se basen en una especificación perfecta. En concreto, atributos del software como mantenibilidad, seguridad o eficiencia no pueden ser especificados explícitamente. Sin embargo, tienen un efecto importante en cómo es percibida la calidad del sistema. Estos atributos se analizarán en la Sección 27.3.

Algunas personas piensan que la calidad puede lograrse definiendo estándares y procedimientos organizacionales de calidad que comprueban si estos estándares son seguidos por el equipo de desarrollo. Su argumento es que los estándares deben encapsular las buenas prácticas, las cuales nos llevan inevitablemente a productos de alta calidad. En la práctica, sin embargo, es más importante la gestión de la calidad que los estándares y la burocracia asociada para asegurar el seguimiento de estos estándares.

Los buenos gestores aspiran a desarrollar una «cultura de la calidad» donde todos seamos responsables de que el desarrollo del producto sea llevado a cabo obteniendo un alto nivel de calidad en éste. Ellos fomentan equipos para responsabilizarse en la calidad de su trabajo y desarrollar nuevas formas para mejorar la calidad. Mientras estándares y procedimientos son las bases de la gestión de la calidad, los gestores de calidad experimentados reconocen que hay aspectos intangibles en la calidad del software (elegancia, legibilidad, etc.) que no puede ser incorporada en los estándares. Ellos ayudan a la gente interesada en estos aspectos intangibles de calidad y fomentan comportamientos profesionales en todos los miembros del equipo.

La gestión formal de la calidad es particularmente importante para equipos que desarrollan sistemas grandes y complejos. La documentación de la calidad es un registro de que es hecho por cada subgrupo en el proyecto. Esto ayuda a la gente a ver qué tareas importantes no deben ser olvidadas o que una parte del equipo no haga suposiciones incorrectas acerca de lo que otros miembros han hecho. La documentación de calidad es también un medio de comunicación sobre el ciclo de vida de un sistema. Ésta permite al grupo responsabilizarse de la evolución del sistema para saber qué ha hecho el equipo de desarrollo.

Para sistemas pequeños, la gestión de calidad es importante todavía, pero se debe adoptar una aproximación más informal. No son tan necesarios los documentos porque el grupo puede comunicarse informalmente. La clave de la calidad en el desarrollo de sistemas pequeños es el establecimiento de cultura de calidad y asegurarse de que todos los miembros del equipo hacen una aproximación positiva a la calidad del software.

La gestión de calidad del software se estructura en tres actividades principales:

1. *Garantía de la calidad.* El establecimiento de un marco de trabajo de procedimientos y estándares organizacionales que conduce a software de alta calidad.
2. *Planificación de la calidad.* La selección de procedimientos y estándares adecuados a partir de este marco de trabajo y la adaptación de éstos para un proyecto software específico.
3. *Control de la calidad.* La definición y fomento de los procesos que garanticen que los procedimientos y estándares para la calidad del proyecto son seguidos por el equipo de desarrollo de software.

La gestión de la calidad provee una comprobación independiente de los procesos de desarrollo software. Los procesos de gestión de la calidad comprueban las entregas del proyecto para asegurarse que concuerdan con los estándares y metas organizacionales (Figura 27.1). El equipo de garantía de calidad debe ser independiente del equipo de desarrollo para que puedan tener una visión objetiva del software. Ellos transmitirán los problemas y las dificultades al gestor principal de la organización.

Un equipo independiente debe ser responsable de la gestión de la calidad y debe informar al gestor del proyecto. El equipo de calidad no está asociado con ningún grupo de desarrollo, sino que tiene la responsabilidad de la gestión de la calidad en toda la organización. La razón de esto es que los gestores del proyecto deben mantener el presupuesto y la agenda. Si aparecen problemas, éstos pueden verse tentados de comprometer la calidad del producto para mantener su agenda. Un equipo independiente de calidad garantiza que los objetivos organizacionales y la calidad no sean comprometidos por consideraciones de presupuesto o agenda.

27.1 Calidad de proceso y producto

Una suposición subyacente de la gestión de calidad es que la calidad del proceso de desarrollo afecta directamente a la calidad de los productos derivados. Esta suposición viene de los sistemas manufactureros donde la calidad del producto está íntimamente ligada al proceso de producción. En un sistema automatizado de producción, el proceso implica configurar y ope-

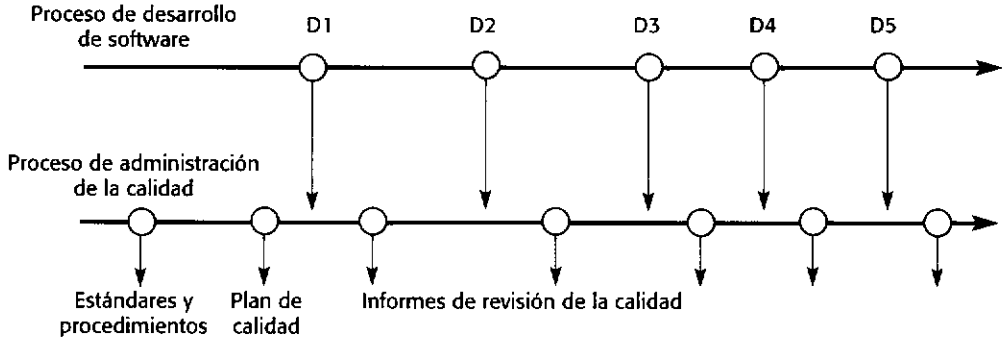


Figura 27.1 La gestión de calidad y el desarrollo de software.

rar la maquinaria asociada al proceso. Si la maquinaria es operada correctamente, la calidad del producto continúa. Se mide la calidad del producto y se cambia el proceso hasta conseguir el nivel de calidad deseado. La Figura 27.2 ilustra esta aproximación basada en proceso para conseguir la calidad del producto.

Hay un vínculo claro entre la calidad del proceso y del producto en producción debido a que el proceso es relativamente fácil de estandarizar y monitorizar.

Cada sistema de producción se calibra, y debe producir una y otra vez productos de alta calidad. Sin embargo, el software no se manufactura, sino que se diseña. El desarrollo de software es un proceso más creativo que mecánico, donde la experiencia y habilidades individuales son importantes. La calidad del producto, sea cual fuere el producto utilizado, también se ve afectada por factores externos, como la novedad de una aplicación o la presión comercial para sacar un producto rápidamente.

En el desarrollo software, por lo tanto, la relación entre la calidad del proceso y la calidad del producto es muy compleja. Es difícil de medir los atributos de la calidad del software, como mantenibilidad, incluso después de utilizar el software durante un largo periodo. En consecuencia, es difícil explicar cómo influyen las características del proceso en estos atributos. Además debido al papel del diseño y la creatividad en el proceso software, no podremos predecir la influencia de los cambios en el proceso en la calidad del producto.

A pesar de ello, la experiencia nos muestra que la calidad del proceso tiene una influencia significativa en la calidad del software. La gestión y mejora de la calidad del proceso debe minimizar los defectos en el software entregado.

La gestión de la calidad del proceso implica:

1. Definir estándares de proceso, como las revisiones a realizar, cuándo llevarlas a cabo, etcétera.
2. Supervisar el proceso de desarrollo para asegurar que se sigan los estándares.
3. Hacer informes del proceso para el gestor del proyecto y para el comprador del software.

Un problema de la garantía de la calidad basada en el proceso es que el equipo de garantía de la calidad (QA) insista en unos estándares de proceso independientemente del tipo de software a desarrollar. Por ejemplo, los estándares de calidad del proceso para sistemas críticos deben requerir una especificación completa y aprobada antes de que comience la implementación. Sin embargo, algunos sistemas críticos pueden necesitar prototipado, lo cual implica empezar a implementar sin una especificación completa. Existen situaciones en las que el equipo de gestión de calidad sugiere que este prototipo no se puede llevar a cabo debido a que su calidad no se puede supervisar. En tales situaciones, el gestor principal debe intervenir para asegurar que el proceso de calidad ayude al desarrollo del producto en lugar de impedirlo.

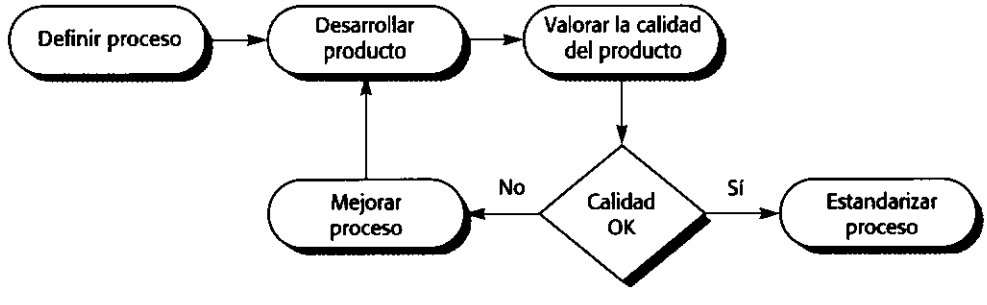


Figura 27.2 Calidad basada en procesos.

27.2 Garantía de la calidad y estándares

La garantía de la calidad es el proceso que define cómo lograr la calidad del software y cómo la organización de desarrollo conoce el nivel de calidad requerido en el software. Como se indicó anteriormente, el proceso QA se ocupa ante todo de definir o seleccionar los estándares que deben de ser aplicados al proceso de desarrollo software o al producto software. Como parte del proceso QA, usted puede seleccionar herramientas y métodos que apoyen estos estándares.

Podemos definir dos tipos de estándares como parte del proceso de garantía de calidad:

1. *Estándares de producto.* Estos estándares se aplican sobre el producto software que se comienza a desarrollar. Incluyen estándares de documentación, como cabecera de comentarios estándar para definición de clases, y estándares de codificación, que definen cómo debe utilizarse el lenguaje de programación.
2. *Estándares de proceso.* Estos estándares definen los procesos que deben seguirse durante el desarrollo del software. Pueden incluir definiciones de procesos de especificación, diseño y validación, así como una descripción de los documentos que deben escribirse en el curso de estos procesos.

Como se propuso en la Sección 27.1, existe una relación muy cercana entre los estándares de producto y los estándares de proceso. Los estándares de producto se aplican a las salidas del proceso software y, en muchos casos, los estándares de proceso incluyen actividades de proceso específicas que garantizan que se sigan los estándares de producto.

Los estándares de software son importantes por varias razones:

1. Están basadas en el conocimiento de la mejor o más apropiada práctica de la empresa. A menudo, este conocimiento sólo se adquiere después de seguir un proceso de prueba y error. Tenerlo constituido en un estándar evita la repetición de errores anteriores. Los estándares captan el conocimiento que es de valor para la organización.
2. Proveen un marco de trabajo alrededor del cual se implementa el proceso de garantía de la calidad. Puesto que los estándares captan las mejores prácticas, el control de la calidad sencillamente asegura que los estándares se siguen adecuadamente.
3. Ayudan a la continuidad cuando una persona continúa el trabajo que llevaba a cabo otra. Los estándares aseguran que todos los ingenieros de una organización adopten las mismas prácticas. En consecuencia, se reduce el esfuerzo de aprendizaje cuando se comienza un nuevo trabajo.

El desarrollo de los estándares de los proyectos de ingeniería del software es un proceso difícil y largo. Organizaciones nacionales e internacionales como el Departamento de Defensa de Estados Unidos, ANSI, BSI, OTAN y el IEEE han estado activas en la producción de los estándares. Estos estándares generales se aplican a una variedad de proyectos. Organizaciones como la OTAN y otras organizaciones de defensa exigen que se sigan sus propios estándares en los contratos de software.

Se han desarrollado estándares nacionales e internacionales para cubrir la terminología, los lenguajes de programación como Java y C++, las notaciones como los símbolos de los diagramas, los procedimientos para derivar y redactar los requerimientos de software, los procedimientos de garantía de calidad, y los procesos de verificación y validación del software (IEEE, 2003).

Los equipos de garantía de calidad que desarrollan los estándares se apoyan en sus estándares organizacionales basados en estándares nacionales e internacionales. Utilizando estos estándares como punto de partida, el equipo de garantía de la calidad debe crear un «manual» de estándares. Éste define los estándares que son apropiados para la organización. La Figura 27.3 muestra ejemplos de estándares que se pueden incluir en ese manual.

Algunas veces, los ingenieros de software consideran a los estándares como burocráticos e irrelevantes para las actividades técnicas de desarrollo de software. Esto es probable sobre todo cuando los estándares requieren llenar formularios tediosos y registrar el trabajo. Aunque por lo general los ingenieros están de acuerdo en que los estándares son necesarios, a menudo tienen buenas razones de por qué dichos estándares no son necesariamente apropiados para su proyecto en particular.

Para evitar estos problemas, los gestores de la calidad que fijan los estándares necesitan estar informados y tomar en consideración los siguientes pasos:

1. Involucrar a los ingenieros de software en el desarrollo de los estándares del proyecto. Así comprenderán la necesidad de diseñar los estándares y se comprometerán con ellos. El documento de los estándares no debe establecer simplemente que se sigan los estándares, sino que deben de incluirse razones de por qué se tomaron decisiones particulares.
2. Revisar y modificar los estándares de forma regular con el fin de reflejar los cambios en la tecnología. Una vez que los estándares se desarrollan, tienden a plasmarse en un manual de estándares de la compañía y a menudo existe cierta reticencia a cambiarlos. Un manual de estándares es esencial, pero debe evolucionar de acuerdo con las circunstancias y la tecnología existentes.
3. Proveer herramientas de software para apoyar los estándares donde sea necesario. Los estándares burocráticos son la causa de muchas quejas debido al trabajo tedioso que se requiere para implementarlos. Si se dispone de una herramienta de apoyo, no se necesitará mucho esfuerzo para seguir los estándares.

Si se impone un proceso no práctico al equipo de desarrollo, los estándares de proceso pueden causar dificultades. Diferentes tipos de software requieren distintos procesos de desarrollo. No existe razón para prescribir una forma particular de trabajo si ésta es inapropiada para un proyecto o para un equipo. Cada gestor de proyecto tiene la facultad de modificar los estándares de proceso de acuerdo con circunstancias particulares. Sin embargo, los estándares relacionados con la calidad del producto y del proceso posterior a la entrega sólo deben cambiarse después de cuidadosas consideraciones.

El gestor del proyecto y el gestor de calidad pueden evitarse los problemas de estándares inapropiados si planean cuidadosamente la calidad. Deben decidir cuáles son los estándares

Formulario para revisión del diseño	Conducto para la revisión del diseño
Estructura del documento de requerimientos	Sometimiento de documentos a CM
Formato del encabezado del método	Proceso de entrega de versiones
Estilo de programación en Java	Proceso de aprobación del plan del proyecto
Formato del plan del proyecto	Proceso de control de cambios
Formulario de petición de cambios	Proceso de registro de pruebas

Figura 27.3 Estándares de producto y de proceso.

del manual que utilizarán sin cambio alguno, cuáles se modificarán y cuáles se dejarán de lado. Pueden tener que crearse nuevos estándares para un requerimiento particular. Por ejemplo, se pueden requerir estándares para la especificación formal si no han sido utilizadas en proyectos anteriores. Conforme el equipo gana experiencia, se deberán modificar y ampliar estos nuevos estándares.

27.2.1 ISO 9000

Un conjunto de estándares internacionales que se puede utilizar en el desarrollo de un sistema de gestión de calidad en todas las industrias es ISO 9000. Los estándares ISO 9000 pueden aplicarse a un amplio abanico de organizaciones desde las de manufactura hasta las de servicios. ISO 9001 es el más general de estos estándares y se aplica en organizaciones interesadas en el proceso de calidad de diseño, desarrollo y mantenimiento de productos. Un documento de ayuda (ISO 9000-3) interpreta ISO 9001 para el desarrollo de software. Existen varios libros que describen el estándar ISO 9001 (Johnson, 1993; Oskarsson y Glass, 1995; Peach, 1996; Bamford y Deibler, 2003).

ISO 9001 no es un estándar específico para desarrollo de software, pero define principios generales que pueden aplicarse al software. El estándar ISO 9001 describe varios aspectos del proceso de calidad y define qué estándares y procedimientos deben existir en una organización. Éstos deben documentarse en un manual de calidad organizacional. La definición del proceso debe incluir una descripción de la documentación requerida, donde se demuestre que los procesos definidos han sido seguidos durante el desarrollo del producto.

Los estándares ISO 9001 no definen los procesos de calidad que deben usarse. De hecho, no restringe los procesos usados en una organización de ninguna forma. Esto permite flexibilidad en todos los sectores industriales e implica que pequeñas compañías puedan tener procesos no burocráticos y que cumplan con la normativa ISO 9000. Sin embargo, esta flexibilidad implica que no podamos hacer suposiciones acerca del parecido o diferencia entre los procesos de compañías que cumplen con la normativa ISO 9000. La Figura 27.4 muestra las áreas que abarca ISO 9001. Este libro no tiene suficiente espacio para exponer este estándar en profundidad. Ince (Ince, 1994) y Oskarsson y Glass (Oskarsson y Glass, 1995) dan detalles de cómo se puede utilizar el estándar para desarrollar procesos de gestión de calidad del software. La Figura 27.5 muestra las relaciones entre ISO 9000, el manual de calidad y los planes de calidad de proyectos particulares. Esta figura se ha creado a partir de un modelo del libro de Ince (Ince, 1994).

Control de productos disconformes	Control de diseño
Manejo, almacenamiento, embalaje y suministro	Compras
Productos suministrados al comprador	Identificación y seguimiento del producto
Control de proceso	Inspección y prueba
Equipo de inspección y prueba	Status de la inspección y las pruebas
Revisión del contrato	Acción correctiva
Control del documento	Registros de calidad
Auditoría de calidad interna	Capacitación
Servicios	Técnicas estadísticas

Figura 27.4 Áreas comprendidas por el modelo ISO 9001 para la garantía de calidad.

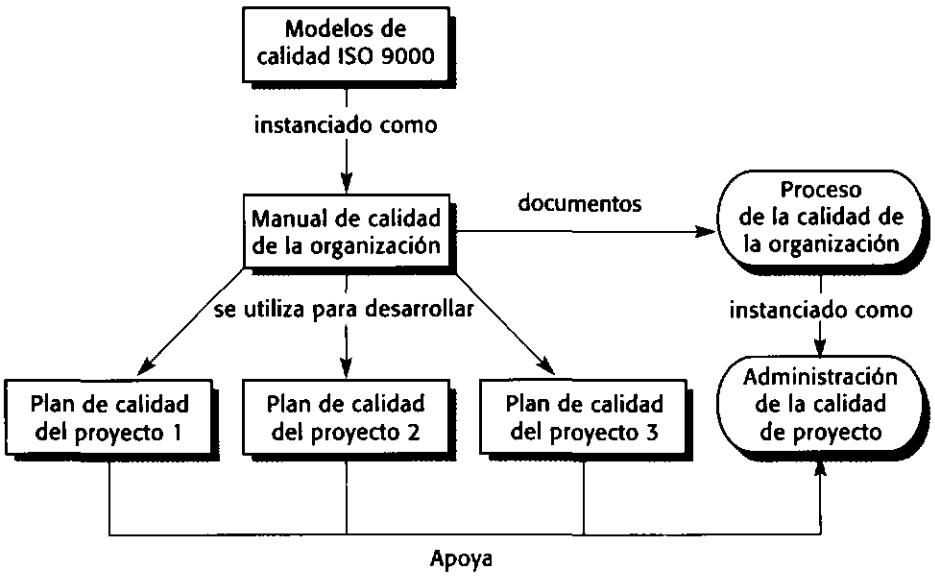


Figura 27.5 ISO 9000 y la gestión de la calidad.

Los procesos de garantía de calidad en una organización se documentan en un manual de calidad que define el proceso de calidad. En algunos países, autoridades acreditadas certifican que el proceso de calidad está reflejado en el manual de calidad y es conforme al estándar ISO 9001. Cada vez más, los clientes buscan el certificado ISO 9000 como indicador de la seriedad con la que el proveedor ve la calidad.

Algunas personas piensan que la certificación ISO 9000 significa que la calidad del software producido por las compañías certificadas será mejor que la del de las compañías no certificadas. Este no es ciertamente el caso. El estándar ISO 9000 se refiere simplemente a la definición de los procedimientos que son utilizados en la compañía y la documentación asociada que muestre que los procesos han sido seguidos. Éste no se ocupa de asegurar que estos procesos sean la mejor práctica, ni asegura la calidad del producto.

Por lo tanto, una compañía puede definir sus procedimientos para la prueba del producto, que realice una prueba incompleta del software. Si estos procedimientos son seguidos y documentados, la compañía podría conseguir el estándar ISO 9001. Mientras esta situación es inverosímil, no hay duda de que algunos estándares de compañías son bastante flojos y hacen sólo una pequeña contribución a la calidad del software.

27.2.2 Estándares de documentación

Los estándares de documentación en un proyecto de software son documentos muy importantes ya que son la única forma tangible de representar al software y su proceso. Los documentos estandarizados tienen una apariencia, estructura y calidad consistentes y, por lo tanto, son más fáciles de leer y de comprender.

Existen tres tipos de estándares de documentación:

1. *Estándares del proceso de documentación.* Definen el proceso a seguir para la producción del documento.
2. *Estándares del documento.* Gobiernan la estructura y presentación de los documentos.
3. *Estándares para el intercambio de documentos.* Aseguran que todas las copias electrónicas de los documentos sean compatibles.

Los estándares del proceso de documentación definen el proceso utilizado para producir los documentos. Esto implica definir los procedimientos involucrados en el desarrollo del documento y las herramientas de software utilizadas. También definen procedimientos de comprobación y refinamiento que aseguren que se produzcan documentos de alta calidad.

Los estándares de calidad del proceso de documentos deben ser flexibles y les debe ser posible ajustarse a todos los tipos de documentos. Para los documentos de trabajo o memorandum, no es necesario comprobar explícitamente la calidad. Sin embargo, si los documentos son documentos formales utilizados para desarrollos posteriores o son documentos para entregar a los clientes, se debe adoptar un proceso formal de calidad. La Figura 27.6 muestra un modelo de uno de los posibles procesos de documentación.

Crear un borrador, comprobarlo, revisarlo y rehacerlo es un proceso iterativo. Éste continúa hasta que se produce un documento de calidad aceptable. El nivel de calidad aceptable depende del tipo de documento y de los lectores potenciales de éste.

Los estándares de documento se aplican a todos documentos producidos en el transcurso del desarrollo de software. Los documentos deben tener un estilo y una apariencia consistentes y los documentos del mismo tipo deben tener una estructura consistente. Aunque los estándares del documento se adapten a las necesidades de un proyecto específico, una buena práctica es que se utilice el mismo estilo en todos documentos producidos por una organización.

Algunos ejemplos de estándares de documentos a desarrollar son:

1. *Estándares de identificación de documentos.* Puesto que los proyectos de sistemas grandes producen cientos de documentos, cada documento debe identificarse de forma única. Para los documentos formales, este identificador es el identificador formal definido por el gestor de configuraciones. Para documentos informales, el estilo del identificador del documento es definido por el gestor del proyecto.
2. *Estándares de la estructura del documento.* Cada clase de documentos producida durante un proyecto de software debe seguir alguna estructura estándar. Los estándares de estructura deben definir las secciones a incluir y especificar las con-

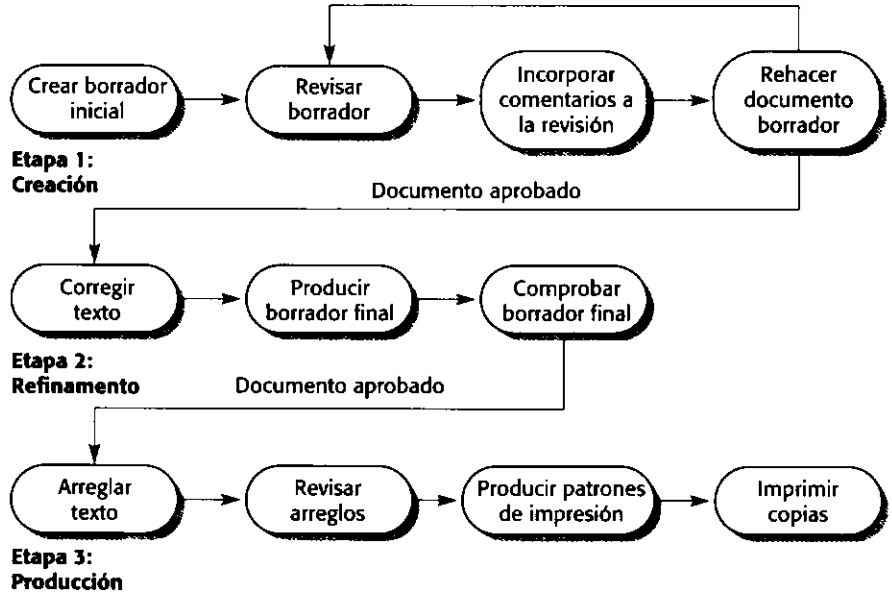


Figura 27.6 Un ejemplo de proceso de producción de documentos que incluye comprobaciones de la calidad.

venciones utilizadas para la numeración de páginas, el encabezado de las páginas y la información de los pies de página, así como la numeración de secciones y subsecciones.

3. *Estándares de presentación de documentos.* Estos estándares definen un «estilo propio» para los documentos y contribuyen notablemente a la consistencia de éstos. Incluyen la definición de tipos de letra y estilos utilizados en el documento, la utilización de logotipos y los nombres de la compañía, la utilización de color para resaltar la estructura del documento, etcétera.
4. *Estándares para actualizar los documentos.* Conforme el documento evoluciona y refleja los cambios en el sistema, se debe utilizar una forma consistente para indicar los cambios en el documento. Se pueden utilizar diversos colores de la portada para indicar una nueva versión del documento y poner marcas en el margen para indicar párrafos modificados o agregados.

Los estándares de intercambio de documentos son importantes debido a que se pueden intercambiar copias electrónicas de los documentos. La utilización de estándares de intercambio permite que los documentos se transfieran electrónicamente y puedan reconstruirse en su forma original.

Suponiendo que la utilización de herramientas estándar es obligatoria en los estándares del proceso, los de intercambio definen las convenciones para emplear estas herramientas. Ejemplos de estándares de intercambio son la utilización de una hoja de estilo estándar cuando se emplea un procesador de textos o las limitaciones en el uso de macros para evitar infecciones de virus. Los estándares de intercambio también delimitan los tipos de letra y los estilos del texto utilizados debido a los diversos recursos de visualización e impresión.

27.3 Planificación de la calidad

La planificación de la calidad es el proceso en el cual se desarrolla un plan de calidad para un proyecto. El plan de calidad define la calidad del software deseado y describe cómo valorar ésta. Por lo tanto, define lo que es software de «alta calidad». Sin esta definición, los diferentes ingenieros pueden trabajar en direcciones opuestas para optimizar los atributos de proyecto.

El plan de calidad selecciona los estándares organizacionales apropiados para un producto y un proceso de desarrollo particulares. Si el proyecto utiliza nuevos métodos y herramientas, se tienen que definir nuevos estándares. Humphrey (Humphrey, 1989), en su libro clásico sobre gestión del software, sugiere una estructura para un plan de calidad. Esta estructura comprende:

1. *Introducción del producto.* Contiene la descripción del producto, el mercado al que se dirige y las expectativas de calidad.
2. *Planes de producto.* Contiene las fechas de terminación del producto y las responsabilidades importantes junto con los planes para la distribución y el servicio.
3. *Descripciones del proceso.* Contiene los procesos de desarrollo y de servicio a utilizar para el desarrollo y administración del producto.
4. *Metas de calidad.* Contiene las metas y planes de calidad para el producto, incluyendo la identificación y justificación de los atributos de calidad importantes del producto.
5. *Riesgos y gestión de riesgos.* Contiene los riesgos clave que podrían afectar a la calidad del producto y las acciones para abordar estos riesgos.

Seguridad	Comprensión	Portabilidad
Protección	Experimentación	Usabilidad
Fiabilidad	Adaptabilidad	Reutilización
Flexibilidad	Modularidad	Eficacia
Robustez	Complejidad	Aprendizaje

Figura 27.7 Atributos de la calidad del software.

Los planes de calidad obviamente difieren dependiendo del tamaño y del tipo de sistema que se desarrolle. Sin embargo, cuando se redactan los planes de calidad, es necesario tratar de mantenerlos lo más compactos posible. Si el documento es muy grande, los ingenieros no lo leerán y esto frustra el propósito de producir un plan de calidad.

Existe una amplia variedad de atributos de calidad del software potenciales (véase la Figura 27.7) a considerar en el proceso de planificación de la calidad. En general, no es posible optimizar todos los atributos para un sistema. El plan de calidad define los atributos de calidad más importantes para el producto a desarrollar. Puede ser que la eficacia sea primordial, por lo que será necesario sacrificar otros factores para alcanzarla. Esto se establece en el plan, y los ingenieros que trabajan en el desarrollo deben cooperar para lograrlo. El plan también define el proceso de evaluación de la calidad. Es una forma estándar de valorar si algún atributo de calidad, como la mantenibilidad o la robustez, está presente en el producto.

27.4 Control de la calidad

El control de la calidad implica vigilar el proceso de desarrollo de software para asegurar que se siguen los procedimientos y los estándares de garantía de calidad. Como ya se indicó en este capítulo (véase la Figura 27.1), en el proceso de control de calidad del proyecto se comprueba que las entregas cumplan los estándares definidos.

Existen dos enfoques complementarios que se utilizan para comprobar la calidad de las entregas de un proyecto:

1. Revisiones de la calidad donde el software, su documentación y los procesos utilizados en su desarrollo son revisados por un grupo de personas. Se encargan de comprobar que se han seguido los estándares del proyecto y el software y los documentos concuerdan con estos estándares. Se toma nota de las desviaciones de los estándares y se comunican al gestor del proyecto.
2. Valoración automática del software en la que el software y los documentos producidos se procesan por algún programa y se comparan con los estándares que se aplican a ese proyecto de desarrollo en particular. Esta valoración automática comprende una medida cuantitativa de algunos atributos del software. La medición y las métricas del software se estudian en la Sección 27.5.

27.4.1 Revisiones de la calidad

Las revisiones son el método más utilizado para validar la calidad del un proceso o de un producto. Involucran a un grupo de personas que examinan todo o parte del proceso software, los sistemas o su documentación asociada para descubrir problemas potenciales. Las conclusio-

nes de la revisión se registran formalmente y se pasan al autor o a quien sea responsable de corregir los problemas descubiertos.

La Figura 27.8 describe brevemente varios tipos de revisiones, entre ellas las revisiones de gestión de la calidad. Las inspecciones ya se trataron en el Capítulo 22. Las revisiones de progreso son parte del proceso de gestión analizado en el Capítulo 5. Los diversos procesos de revisión tienen muchas cosas en común, y en el Capítulo 22 se describió el proceso para establecer una revisión.

El equipo de revisiones debe tener un núcleo de tres o cuatro personas como revisores principales. Uno debe ser el diseñador principal, el cual tendrá la responsabilidad de tomar las decisiones técnicas. Los revisores principales pueden invitar a otros miembros del proyecto, como a los diseñadores de los subsistemas, para que colaboren en la revisión. Ellos no se involucran en la revisión de todo el documento. Más bien, se concentran en aquellas partes que afectan a su trabajo. De forma alternativa, el equipo de revisión hace circular el documento a revisar y solicita comentarios escritos de otros miembros del proyecto.

Los documentos a revisar deben distribuirse con anterioridad a la revisión para dar tiempo a los revisores a que los lean y los comprendan. Aunque esto puede interrumpir el proceso de desarrollo, la revisión no es eficaz si el equipo de revisión no comprende adecuadamente los documentos antes de que tenga lugar la revisión.

La revisión misma es relativamente corta (dos horas a lo más). El autor del documento en revisión debe seguir el documento junto con el equipo de revisión. Un miembro del equipo preside la revisión y otro registra formalmente todas las decisiones de la revisión. Durante ésta, el presidente es responsable de asegurar que se consideren todos los comentarios escritos. El responsable de la revisión firmará el registro de la reunión, donde aparecerán todos los comentarios y las decisiones tomadas. Este registro pasará a formar parte de la documentación formal del proyecto. Si sólo se descubren problemas menores, no es necesaria ninguna revisión adicional. El presidente es responsable de asegurar que se hagan todos los cambios requeridos. Si se requieren cambios importantes, habrá que hacer un seguimiento posterior de la revisión.

27.5 Medición y métricas del software

Las revisiones de la calidad son caras, consumen tiempo e inevitablemente retrasan la entrega del software. Idealmente, sería posible acelerar el proceso de revisión utilizando herramientas que procesaran el diseño del software o el programa, e hiciesen valoraciones auto-

Inspecciones de diseño o programas	**Detectar errores finos en los requerimientos, el diseño o el código. La revisión es conducida por una lista de verificación de los posibles errores.**
Revisiones del progreso	**Proveer información del progreso del proyecto útil para su gestión. Ésta es una revisión tanto del proceso como del producto y se refiere a costes, duración y planificación.**
Revisiones de la calidad	**Llevar a cabo un análisis técnico de los componentes del producto o documentación para encontrar diferencias entre la especificación y el diseño del componente, código y documentación, y para asegurar que se siguen los estándares de calidad definidos.**

Figura 27.8
Tipos de revisión.

máticas de la calidad del software. Estas valoraciones permiten comprobar que el software tiene el umbral de calidad requerido, y destacar las partes en las cuales no se ha alcanzado para revisarlas.

La medición del software se refiere a derivar un valor numérico desde algún atributo del software o del proceso software. Comparando estos valores entre sí y con los estándares aplicados en la organización, es posible sacar conclusiones de la calidad del software o de los procesos para desarrollarlo. Por ejemplo, supongamos que una organización hace planes para introducir una nueva herramienta de prueba de software. Antes de introducir la herramienta, se registra el número de defectos descubiertos en el software en un tiempo dado; después de introducir la herramienta, se repite este proceso. Si se descubren más defectos en la misma cantidad de tiempo después de introducir la herramienta, entonces parecería que provee una información útil para el proceso de validación del software.

Las mediciones del software pueden utilizarse para:

1. *Hacer predicciones generales acerca del sistema.* Haciendo mediciones de las características de los componentes del sistema y reuniendo éstas, podremos derivar una estimación general de algunos atributos del sistema, como el número de fallos.
2. *Identificar componentes anómalos.* Mediante las mediciones podemos identificar los componentes que se salgan de lo normal. Por ejemplo, podemos medir los componentes para identificar los de complejidades más altas, los cuales suponemos que serán los que tengan más errores, para centrarnos en ellos en el proceso de revisión.

Una métrica de software es cualquier tipo de medida relacionada con un sistema, proceso o documentación de software. Algunos ejemplos son las medidas que se utilizan para calcular el tamaño de un producto en líneas de código; el índice de Fog (Gunning, 1962), que mide la claridad de un párrafo en un texto; el número de fallos encontrados en un producto software entregado; y el número de personas/día requeridas para desarrollar un componente del sistema.

Algunas compañías como Hewlett-Packard (Grady, 1993), AT&T (Barnard y Price, 1994) y Nokia (Kilpi, 2001) han introducido programas de métricas que recogen medidas en sus procesos de gestión de calidad. El enfoque fue recoger medidas de los defectos en los programas y en los procesos de verificación y de validación. Offen y Jeffrey (Offen y Jeffrey, 1997) y Hall y Fenton (Hall y Fenton, 1997) estudian la introducción de tales programas en la industria. El manual ami (aplicación de métricas en la industria) (Pulford *et al.*, 1996) da consejos detallados sobre la medición para la mejora de procesos.

Sin embargo, pocas compañías utilizan sistemáticamente métricas para valorar la calidad del software. Una razón de esto es que, en mucha compañías, los procesos de software están definidos y controlados, y no son lo suficientemente maduros para utilizar medidas. Otra razón es que no existen estándares para las métricas y, por lo tanto, las herramientas para recogida y análisis de datos son muy limitadas. Muchas compañías no estarán dispuestas a introducir mediciones hasta que estén disponibles tales estándares y herramientas.

Las métricas son de control o de predicción. Ambas pueden influir en la toma de decisiones de gestión como muestra la Figura 27.9. Las métricas de control suelen estar asociadas con los procesos, mientras que las métricas de predicción lo están a los productos. Ejemplos de las métricas de control o de procesos son el esfuerzo y el tiempo promedio requeridos para reparar los defectos encontrados. Ejemplos de métricas de predicción son la complejidad ciclomática de un módulo, la longitud media de los identificadores de un programa, y el número de atributos y operaciones asociadas con los objetos de un diseño. En este capítulo, tratare-

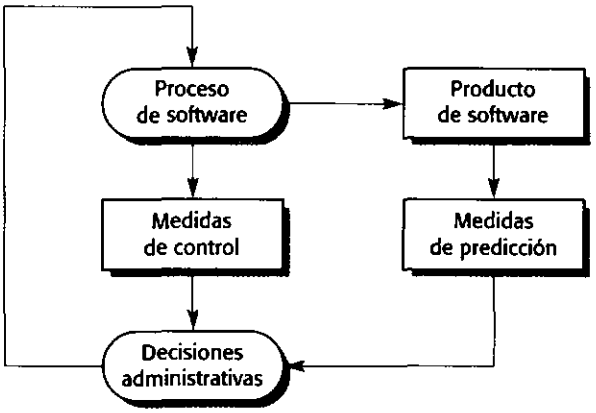


Figura 27.9 Métricas de predicción y de control.

mos las métricas de predicción, ya que nos centraremos en las medidas para predecir la calidad del producto. En el Capítulo 28 veremos las métricas de control.

Frecuentemente, es imposible medir los atributos de calidad del software directamente. Los atributos de calidad como la mantenibilidad, la comprensión y la usabilidad son atributos externos que nos dicen cómo ven el software los desarrolladores y los usuarios. Éstos se ven afectados por diversos factores y no existe un camino simple para medirlos. Más bien es necesario medir atributos internos del software (como su tamaño) y suponer que existe una relación entre lo que queremos medir y lo que queremos saber. De forma ideal, existe una relación clara y válida entre los atributos internos y externos del software.

La Figura 27.10 muestra algunos atributos externos de la calidad que nos serán interesantes y los atributos internos relativos a ellos. El diagrama sugiere que existe una relación entre los atributos externos y los internos, pero no dice qué relación es. Para que la medida del atributo interno sea un indicador útil de la característica externa, se deben cumplir tres condiciones:

1. El atributo interno debe medirse de forma precisa.
2. Debe existir una relación entre lo que se puede medir y el atributo de comportamiento externo.
3. Esta relación se comprende, ha sido validada y se puede expresar en términos de una fórmula o modelo.

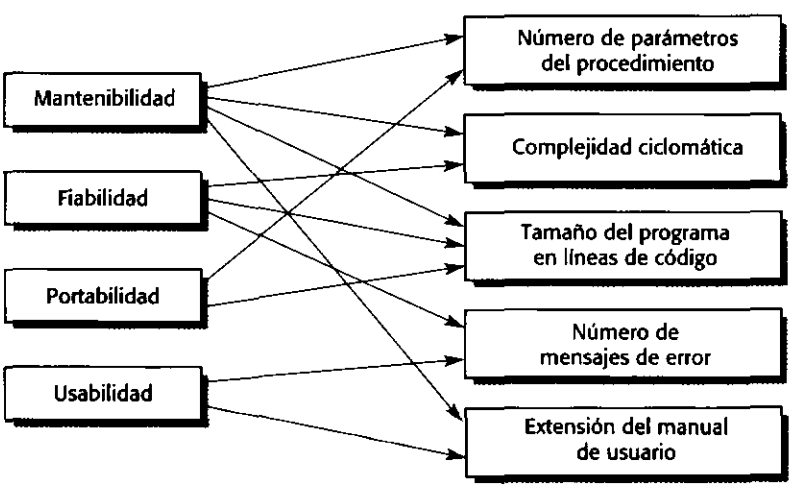


Figura 27.10 Relaciones entre los atributos internos y externos del software.

La formulación del modelo comprende identificar la forma funcional del modelo (lineal, exponencial, etc.) a través del análisis de los datos recogidos, identificar los parámetros que se van a incluir en el modelo y calibrarlos utilizando datos existentes. El desarrollo del modelo, si se confía en él, requiere que se tenga experiencia en técnicas estadísticas e, idealmente, alguien con experiencia y conocimientos debe definir el modelo.

27.5.1 El proceso de medición

La Figura 27.11 muestra el proceso de medición del software dentro de un proceso de control de calidad. Cada uno de los componentes del sistema se analiza por separado y los diversos valores de las métricas se comparan entre sí y, quizás, con los datos históricos de medición recogidos en los proyectos previos. Las medidas anómalas se utilizan para centrar el esfuerzo de garantía de calidad en los componentes que tienen problemas de calidad.

Los pasos clave en este proceso son:

1. *Seleccionar las medidas a realizar.* Se deben formular las preguntas que la medición intenta responder y definir las mediciones requeridas para resolver estas preguntas. No se recogen las mediciones que no están relacionadas de forma directa con estas preguntas. El paradigma GQM (Goal-Question-Metric) de Basili (Basili y Rombach, 1988), que se analiza en el capítulo siguiente, es un buen enfoque a utilizar cuando se decide qué datos se deben recolectar.
2. *Seleccionar los componentes a evaluar.* No es necesario o deseable estimar los valores de las métricas de todos los componentes de un sistema software. En algunos casos, para la medición se elige un conjunto representativo de componentes. En otros, se evalúan los componentes particularmente críticos como son los fundamentales que se utilizan de forma constante.
3. *Medir las características de los componentes.* Se miden los componentes seleccionados y se calculan los valores de las métricas. Normalmente, esto comprende procesar la representación del componente (diseño, código, etc.) utilizando una herramienta de recogida de datos. Esta herramienta se puede crear de forma especial o puede estar incorporada en las herramientas CASE utilizadas en la organización.
4. *Identificar las mediciones anómalas.* Una vez que se obtienen las mediciones de los componentes, se comparan entre sí y con las mediciones previas registradas en una base de datos de mediciones. Se deben observar los valores más altos y los más bajos de cada métrica, puesto que éstos sugieren que puede haber problemas con los componentes que exhiben estos valores.
5. *Analizar los componentes anómalos.* Una vez identificados los componentes con valores anómalos para las métricas particulares, se examinan estos componentes para decidir

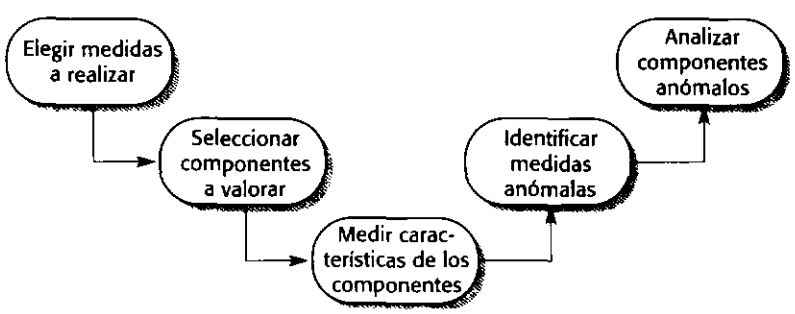


Figura 27.11 falta Falta.

si los valores de la métrica indican que la calidad del componente está en peligro. Los valores de la métrica anómalos para la complejidad (por ejemplo) no significan necesariamente que el componente tenga una calidad deficiente. Puede haber alguna otra razón para ese valor alto y ello no significa que existan problemas con la calidad del producto.

Los datos recogidos se mantienen como un recurso organizacional y deben conservarse registros históricos de todos los proyectos aun cuando los datos no se hayan utilizado durante en un proyecto particular. Una vez que se haya creado una base de datos suficientemente grande de mediciones, se hace la comparación de los proyectos, y las métricas específicas se refinan de acuerdo con las necesidades organizacionales.

27.5.2 Métricas de producto

Las métricas de producto se refieren a las características del software mismo. Desafortunadamente, las características del software que se miden fácilmente, como el tamaño y la complejidad ciclomática, no tienen una relación clara y consistente con los atributos de calidad como la compresión y la mantenibilidad. Las relaciones varían dependiendo de los procesos, la tecnología y el tipo de sistemas a desarrollar. Las organizaciones interesadas en la medición tienen que construir una base de datos históricos. Después, ésta se puede utilizar para descubrir cómo se relacionan los atributos del producto de software con la calidad en la que está interesada la organización.

Las métricas del producto se dividen en dos clases:

1. *Las métricas dinámicas*, que son recogidas por las mediciones hechas en un programa en ejecución.
2. *Las métricas estáticas*, que son recogidas por las mediciones hechas en las representaciones del sistema como el diseño, el programa o la documentación.

Estas diferentes métricas están relacionadas con diversos atributos de calidad. Las métricas dinámicas ayudan a valorar la eficiencia y la fiabilidad de un programa. Las métricas estáticas ayudan a valorar la complejidad, la comprensión y la mantenibilidad de un sistema de software.

Las métricas dinámicas por lo general están relacionadas de forma cercana con los atributos de calidad del software. Es relativamente fácil medir el tiempo de ejecución requerido por funciones particulares y estimar el tiempo requerido para iniciar un sistema. Esto se relaciona directamente con la eficiencia del sistema. De forma similar, se puede registrar el número y el tipo de caídas del sistema y relacionarlos directamente con la fiabilidad del software como se indicó en el Capítulo 24.

Las métricas estáticas, por otro lado, tienen una relación indirecta con los atributos de calidad. Se han propuesto muchas de estas métricas y se han llevado a cabo experimentos para tratar de derivar y validar las relaciones entre estas métricas y la complejidad, la comprensión y la mantenibilidad del sistema. La Figura 27.12 describe varias métricas estáticas utilizadas para valorar los atributos de calidad. De éstas, parece ser que los predictores más confiables de la comprensión, la complejidad y mantenibilidad del sistema son la longitud del programa o del componente y la complejidad ciclomática.

Las métricas de la Figura 27.12 son genéricas, pero se han propuesto métricas específicas para la programación orientada a objetos (Figura 27.13). Las métricas orientadas a objetos más conocidas fueron propuestas por Chidamber y Kemerer (Chidamber y Kemerer, 1994) y hay herramientas para recoger estas métricas. Algunas de estas métricas han derivado de las antiguas métricas de la Figura 27.12, pero otras son métricas únicamente para sistemas orien-

Fan-in/Fan-out	Fan-in es una medida del número de funciones o métodos que llaman a otra función o método (por ejemplo, X). Fan-out es el número de funciones que son llamadas por una función X. Un valor alto de fan significa que X está fuertemente acoplada al resto del diseño y que los cambios en X tendrán muchos efectos importantes. Un valor alto de fan-out sugiere que la complejidad de X podría ser alta debido a la complejidad de la lógica de control necesaria para coordinar los componentes llamados.
Longitud del código	Ésta es una medida del tamaño del programa. Generalmente, cuanto más grande sea el tamaño del código de un componente, más complejo y susceptible de errores será el componente. La longitud del código ha mostrado ser la métrica más fiable para predecir errores en los componentes.
Complejidad ciclomática	Ésta es una medida de la complejidad del control de un programa. Esta complejidad del control está relacionada con la comprensión del programa. El cálculo de la complejidad ciclomática se trata en el Capítulo 22.
Longitud de los identificadores	Es una medida de la longitud promedio de los diferentes identificadores en un programa. Cuanto más grande sea la longitud de los identificadores, más probable será que tengan significado; por lo tanto, el programa será más comprensible.
Profundidad del anidamiento de las condicionales	Ésta es una medida de la profundidad de anidamiento de las instrucciones condicionales «if» en un programa. Muchas condiciones anidadas son difíciles de comprender y son potencialmente susceptibles de errores.
Índice de Fog	Ésta es una medida de la longitud promedio de las palabras y las frases en los documentos. Cuanto más grande sea el índice de Fog, el documento será más difícil de comprender.

Figura 27.12 Métricas estáticas de producto software.

tados a objeto. El-Amam (El-Amam, 2001), en una excelente revisión de las métricas orientadas a objetos, expone algunos de estos estudios y concluye que todavía no tenemos suficientes evidencias para entender cómo se relacionan las métricas orientas a objeto con los atributos de calidad externos del software.

Las métricas relevantes dependen del proyecto, las metas del equipo de gestión de calidad y el tipo de software a desarrollar. Todas las métricas mostradas en las Figuras 27.12 y 27.13

Profundidad del árbol de herencia	Ésta representa el número de niveles discretos en el árbol de herencia donde las subclases heredan atributos y operaciones (métodos) de las superclases. Cuanto más profundo sea el árbol de herencia, más complejo será el diseño. Muchas clases de objetos distintas tienen que comprenderse para conocer las clases de objetos en las hojas del árbol.
Método Fan-in/Fan-out	Está directamente relacionada fan-in y fan-out, como se describió antes, y significa esencialmente lo mismo. Sin embargo, es conveniente hacer una distinción entre las llamadas provenientes de otros métodos dentro del objeto y las llamadas provenientes de los métodos externos.
Métodos pesados por clase	Éste es el número de métodos incluidos en una clase con su correspondientes pesos, que vendrán dados por la complejidad de cada método. Por lo tanto, un método sencillo tiene una complejidad de 1 y un método grande y complejo un valor mucho más grande. Cuanto más grande sea el valor de esta métrica, la clase será más compleja. Los objetos complejos tienden a ser más difíciles de comprender. No son lógicamente cohesivos, por lo que no se pueden reutilizar efectivamente como superclases en un árbol de herencia.
Número de operaciones sobrescritas	Éste es el número de operaciones en una superclase que se anulan en una subclase. Un valor alto para esta métrica indica que la superclase utilizada no es una madre adecuada para la subclase.

Figura 27.13 Métricas orientadas a objetos.

pueden semos útiles en alguna situación. De la misma manera, no serán apropiadas que en otras circunstancias. Cuando se introduce una medida de software como parte del proceso de gestión de calidad, las organizaciones deben hacer experimentos para descubrir que métricas son más apropiadas para sus necesidades.

27.5.3 Análisis de las mediciones

Uno de los problemas con la recogida de datos cuantitativos en el software y en los proyectos de software es comprender lo que significan realmente los datos. Es fácil malinterpretar los datos y hacer inferencias incorrectas. Las mediciones se deben analizar cuidadosamente para comprender lo que realmente significan.

Para ilustrar que los datos tomados se pueden interpretar de formas diferentes, consideremos el siguiente escenario. Para hacer más fácil su comprensión, se ha usado una métrica de producto en lugar de una de proceso, pero el mensaje final siempre es el mismo; las razones del cambio frecuentemente son difíciles de comprender.

Un gestor decide supervisar el número de peticiones de cambios solicitados por los clientes basándose en la suposición de que existe una relación entre estas peticiones de cambios y la utilidad y conveniencia del producto. Cuanto más alto es el número de peticiones de cambios, el software cumple menos las necesidades del usuario.

Procesar las peticiones de cambios y cambiar el software es costoso. Por lo tanto, la organización decide modificar su proceso para incrementar la satisfacción del cliente y, al mismo tiempo, reducir los costes del cambio. Se pretende que los cambios en el proceso den como resultado mejores productos y menos peticiones de cambios.

Los cambios en el proceso comienzan involucrando más al cliente en el proceso de diseño de software. Se introducen las pruebas beta para todos los productos, y las modificaciones solicitadas por los clientes se incorporan en el producto a entregar. Se entregan las nuevas versiones de los productos desarrollados con este proceso modificado. En algunos casos, el número de peticiones de cambios se reduce; en otros, se incrementa. El gestor se queda perplejo y no puede valorar los efectos de los cambios del proceso en la calidad del producto.

Para comprender por qué pasan estas cosas, se tiene que comprender por qué se hacen las peticiones de cambio. Una razón es que el software entregado no haga lo que los clientes quieren que haga. Otra posibilidad es que el software sea muy bueno y se utilice amplia y frecuentemente, algunas veces para propósitos para los que no se diseñó originalmente. Puesto que existen muchas personas que lo utilizan, es natural que se generen más peticiones de cambios.

Una tercera posibilidad es que la compañía desarrolladora del software sea sensible a las peticiones de cambio de los clientes. Por lo tanto, los clientes se sienten satisfechos con el servicio que reciben. Generan muchas peticiones de cambios debido a que conocen que esas peticiones se considerarán seriamente. Sus sugerencias probablemente se incorporarán en las versiones posteriores del software.

El número de peticiones de cambios podría decrecer debido a que los cambios en el proceso han sido eficaces y lo han hecho más utilizable y conveniente. De forma alternativa, este número podría haber decrecido debido a que el producto ha perdido mercado con respecto a su producto rival. En consecuencia, existen menos usuarios del producto. El número de peticiones de cambios podría incrementarse debido a que existen más usuarios, debido a que los procesos de pruebas beta han convencido a los usuarios de que la compañía desea hacer cambios o debido a que los sitios de pruebas beta no son típicos o no se utilizan mucho en el programa.

Para analizar los datos de las peticiones de cambios, no basta con conocer el número de peticiones de cambios. Es necesario conocer quién hizo la petición, cómo utiliza el software

y por qué hizo la petición. Es necesario saber si existen factores externos, como las modificaciones en el procedimiento de petición de cambios o cambios en el mercado, que podrían tener algún efecto. Con esta información, es posible descubrir si los cambios del proceso han sido eficaces para incrementar la calidad del producto.

Esto ilustra que la interpretación de datos cuantitativos de un producto o proceso es un proceso subjetivo. Los procesos y productos para medir no están aislados de su entorno y los cambios en ese entorno invalidan las comparaciones de los datos. Los datos cuantitativos de las actividades humanas no siempre pueden tomarse como valores de entrada. Las razones subyacentes al valor medido deben investigarse.

PUNTOS CLAVE

- **La gestión de la calidad del software permite señalar si éste tiene un escaso número de defectos y si alcanza los estándares requeridos de mantenibilidad, fiabilidad, portabilidad, etcétera. Las actividades de la gestión de la calidad comprenden la garantía de la calidad que establece los estándares para el desarrollo de software, la planificación de la calidad y el control de la calidad que comprueba el software con respecto a los estándares definidos.**
- **Un manual de calidad organizacional debe documentar un conjunto de procedimientos de garantía de la calidad. Éste puede basarse en los modelos genéricos sugeridos en los estándares ISO 9000.**
- **Los estándares de software son importantes para garantizar la calidad puesto que representan una identificación de las «mejores prácticas». El proceso de control de calidad implica comprobar que el proceso del software y el software a desarrollar concuerdan con estos estándares.**
- **Las revisiones de los productos a entregar por el proceso del software incumben a un equipo de personas los cuales comprobarán que se han seguido los estándares de calidad. Las revisiones son la técnica más utilizada para valorar la calidad.**
- **Las mediciones de software se utilizan para recoger datos cuantitativos acerca del software y sus procesos. Los valores de las métricas de software recogidas se utilizan para hacer inferencias de la calidad del producto y del proceso.**
- **Las métricas de calidad del producto son de gran valor para resaltar los componentes anómalos que tienen problemas de calidad. Estos componentes se deberán analizar con más detalle.**
- **No existen métricas de software estandarizadas y aplicables universalmente. Las organizaciones deben seleccionar métricas y analizar mediciones basadas en el conocimiento y circunstancias locales.**

LECTURAS ADICIONALES

Software Quality Assurance: From Theory to Implementation. Una excelente y actualizada mirada a los principios y prácticas de la garantía de la calidad. Incluye un estudio sobre los estándares ISO 9001. (D. Galin, 2004, Addison-WesLey.)

Metrics and Models for Software Quality Engineering, 2nd ed. Es una exposición fácil de comprender acerca de las métricas que incluyen el proceso software. Contiene las bases matemáticas necesarias para el desarrollo y la comprensión de los modelos en los que se basan las mediciones del software. (S. H. Kan, 2003, Addison-Wesley.)

«Making sense of measurement for small organisations». Éste es un artículo interesante acerca de las aplicaciones prácticas de las métricas. Señala que la utilización de las métricas tiene que considerar su contexto. (K. Kautz, *IEEE Software*, marzo-abril de 1999.)

A Quantitative Approach to Software Management: The ami Handbook. Es una excelente guía acerca de cómo introducir un programa de medición y la utilización de los resultados para mejorar el proceso. Desafortunadamente, es difícil de encontrar. (K. Pulford *et al.*, 1996, Addison-Wesley.)

EJERCICIOS

27.1 Explique por qué un proceso del software de alta calidad debe conducir a productos de software de alta calidad. Comente los posibles problemas con este sistema de gestión de la calidad.

27.2 Explique cómo deben utilizarse los estándares para captar la sabiduría organizacional acerca de sus métodos de desarrollo de software efectivos. Indica cuatro tipos de conocimiento que deben recoger los estándares organizacionales.

27.3 Comente la valoración de la calidad del software de acuerdo con los atributos de calidad mostrados en la Figura 27.7. Explique cómo se podría evaluar cada uno de los atributos.

27.4 Diseñe un formulario electrónico que se pueda utilizar para registrar los comentarios de la revisión y para hacer comentarios a los revisores por medio del correo electrónico.

27.5 Describa brevemente los posibles estándares que se podrían utilizar para:

- Estructuras de control en C, C++ o Java
- Crear informes para un proyecto de fin de curso de una universidad
- El proceso de realizar y aprobar cambios a un programa (véase el Capítulo 29)
- El proceso de comprar e instalar una computadora.

27.6 Suponga que trabaja en una organización que desarrolla productos de bases de datos para sistemas de microcomputadoras. Esta organización está interesada en cuantificar su desarrollo de software. Escriba un informe que indique las métricas apropiadas y cómo recoger estas métricas.

27.7 Explique por qué las métricas de diseño de software, por sí mismas, son un método inadecuado para predecir la calidad del diseño.

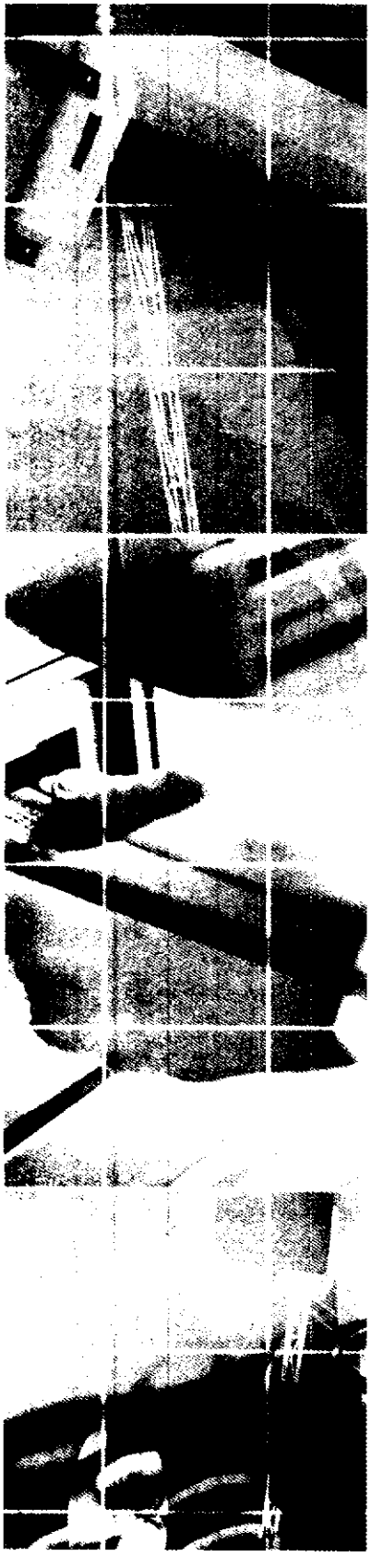
27.8 Consulte la literatura y encuentre otras métricas de calidad para el diseño sugeridas, además de las comentadas aquí. Considere estas métricas en detalle y evalúe si conducen a valores reales.

27.9 Explique por qué es difícil validar las relaciones entre los atributos internos del software, como la complejidad ciclomática, y los atributos externos, como la mantenibilidad.

27.10 ¿Cómo podrían utilizarse las mediciones automáticas en programación extrema? (véase el Capítulo 17).

27.11 ¿Frenan los estándares software la innovación tecnológica?

27.12 Un colega que es un buen programador produce software con un número bajo de defectos, pero frecuentemente prescinde de los estándares de calidad organizacionales. ¿Cómo deberían los administradores de la organización reaccionar a este comportamiento?



28 Mejora de procesos

Objetivos

El objetivo de este capítulo es explicar cómo se pueden mejorar los procesos del software para obtener el mejor producto. Cuando termine de leer este capitulo:

- comprenderá los principios de mejora de procesos del software y por qué vale la pena esta mejora;
- comprenderá cómo los factores del proceso del software influyen en la calidad del software y en la productividad de los desarrolladores;
- será capaz de desarrollar modelos simples de procesos de desarrollo de software;
- entenderá las nociones de capacidad y madurez de proceso, y el marco general del modelo CMMI para la mejora de procesos.

Contenidos

Como vimos en el Capítulo 27, existe un vínculo estrecho entre la calidad del proceso de desarrollo y la calidad de los productos desarrollados utilizando dicho proceso. En consecuencia, muchas compañías de ingeniería del software han tomado el camino de la mejora de los procesos del software para mejorar su software. La mejora de procesos significa entender los procesos existentes y cambiarlos para mejorar la calidad del producto y/o reducir los costes y el tiempo de desarrollo. Mucha de la literatura sobre mejora de procesos se ha centrado en la optimización de los procesos para mejorar la calidad del producto y, en particular, en reducir el número de defectos en el software entregado. Una vez que se logra esto, las metas principales son la reducción de costes y tiempo de desarrollo.

Los procesos del software son intrínsecamente complejos y comprenden un gran número de actividades. Como los productos, los procesos también tienen atributos o características como se puede ver en la Figura 28.1. No es posible hacer mejoras de procesos que optimicen todos los atributos del proceso de forma simultánea. Por ejemplo, si se requiere un proceso de desarrollo rápido, entonces es necesario reducir la visibilidad del proyecto. Hacer un proceso visible significa producir documentos a intervalos regulares. Y esto hace que inevitablemente el proceso sea lento.

La mejora de procesos no significa simplemente adoptar métodos o herramientas particulares o utilizar algún modelo de un proceso utilizado en lugar de otro. Aunque las organizaciones que desarrollan el mismo tipo de software claramente tienen mucho en común, siempre existen factores organizacionales particulares, procedimientos y estándares que influyen en el proceso. Raramente se tendrá éxito cambiando simplemente a un proceso utilizado en otro lugar. Debe verse la mejora de procesos como una actividad específica de una organización o de parte de ella si es una organización grande.

La mejora de procesos es una actividad cíclica, tal como muestra la Figura 28.2. Y tiene tres estados principales:

1. Proceso de medición de los atributos del proyecto actual o del producto. El objetivo es mejorar las mediciones de acuerdo con las metas de la organización involucrada en el proceso de mejora.

Comprensión	**¿Hasta qué punto se define completamente el proceso y cómo de fácil es comprender su definición?**
Visibilidad	**¿Las actividades del proceso culminan en resultados claros de forma que el progreso del proceso es visible externamente?**
Apoyo	**¿Hasta qué punto las actividades del proceso pueden apoyarse en herramientas case?**
Aceptación	**¿El proceso definido es aceptable y utilizable por los ingenieros responsables de producir el software?**
Fiabilidad	**¿El proceso diseñado es de tal forma que los errores del proceso se evitan o identifican antes de que se conviertan en errores de producto?**
Robustez	**¿Puede continuar el proceso a pesar de los problemas inesperados?**
Mantenibilidad	**¿Puede evolucionar el proceso para reflejar los requerimientos organizacionales cambiantes o las mejoras identificadas del proceso?**
Rapidez	**¿Cómo de rápido se puede completar el proceso de construcción de un sistema a partir de una especificación dada?**

Figura 28.1 Características de los procesos.

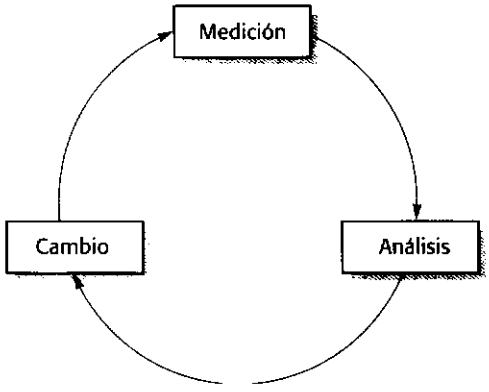


Figura 28.2
El ciclo de mejora de procesos.

2. Proceso de análisis. El proceso actual es valorado, y se identifican puntos flacos y cuellos de botella. En esta etapa se suelen desarrollar los procesos que describen los modelos de proceso.
3. Introducción de los cambios del proceso identificados en el análisis.

Cada una de estas secciones es cubierta en secciones separadas de este capítulo. Cada etapa del proceso puede durar varios meses. La mejora de procesos es una actividad a largo plazo. Es una actividad continua, en la que se introducen nuevos procesos, el entorno de negocio cambia y los procesos por sí mismos evolucionan para tener en cuenta estos cambios.

28.1 Calidad de producto y de proceso

La mejora de procesos está basada en la suposición de que la calidad del proceso de desarrollo es crítica para la calidad del producto. Estas nociones de mejora de procesos provienen del ingeniero estadounidense W. E. Deming, quien trabajó en la industria japonesa después de la Segunda Guerra Mundial para mejorar la calidad. La industria japonesa había estado dedicada al proceso de mejora continua de procesos durante muchos años. Esto contribuyó enormemente a la calidad de los bienes manufacturados japoneses.

Deming y otros introdujeron la idea de control estadístico de la calidad. Ésta se basa en medir el número de defectos en los productos y relacionar estos defectos con el proceso. Éste se mejora con el propósito de reducir el número de defectos en el producto y hasta que sea repetible; esto es, hasta que los resultados del proceso sean predecibles y el número de defectos se reduzca. Después se estandariza e inicia un ciclo de mejoras de calidad.

Humphrey, en su libro fundamental sobre gestión de procesos (Humphrey, 1988), comenta que las mismas técnicas pueden ser aplicadas a la ingeniería del software. Él escribió:

> *W. E. Deming, en su trabajo con la industria japonesa después de la Segunda Guerra Mundial, aplicó los conceptos de control estadístico de procesos a la industria. Aunque hay diferencias importantes, estos conceptos son aplicables al software al igual que si fuesen coches, cámaras, relojes o acero.*

Aunque hay claras similitudes, aquí mostramos desacuerdo con Humphrey en que los resultados de la ingeniería de la industria manufacturera puedan ser transferidos directamente a la ingeniería del software. Cuando existe manufactura, la relación entre proceso y producto es obvia. Mejorar un proceso de tal forma que se eliminen los defectos conduce a mejores

productos. Este vínculo es menos obvio cuando el producto es intangible y dependiente, y de cierta forma podemos decir que los procesos intelectuales no se pueden automatizar. La calidad del software no depende de un proceso de manufactura sino de un proceso de diseño en el que las capacidades del individuo son importantes. Para algunas clases de productos, el proceso utilizado es el determinante más importante de la calidad del producto. Sin embargo, para aplicaciones innovadoras en particular, la gente involucrada en el proceso es más importante que el proceso utilizado.

Para los productos software, o para cualquier otro producto intelectual como libros o películas donde la calidad del producto depende del diseño, existen cuatro factores que afectan a la calidad del producto. Éstos se muestran en la Figura 28.3.

La influencia de cada uno de estos factores depende del tamaño y del tipo de proyecto. Para sistemas muy grandes compuestos de subsistemas independientes, desarrollados por equipos que pueden trabajar en diferentes localizaciones, el determinante principal de la calidad del producto es el proceso del software. Los problemas principales con los proyectos grandes son la integración, la gestión y las comunicaciones. Por lo general existe una mezcla de habilidades y de experiencia en los miembros del equipo y, puesto que el proceso de desarrollo requiere varios años, el equipo de desarrollo es volátil. Puede cambiar totalmente en el tiempo de vida del proyecto. Por lo tanto, los individuos con talento y habilidades particulares por lo general no tienen un efecto dominante en el tiempo de vida del proyecto.

Sin embargo, para proyectos pequeños donde existen únicamente unos pocos miembros, la calidad del equipo de desarrollo es más importante que el proceso de desarrollo utilizado. Si el equipo tiene un nivel alto de habilidad y experiencia, la calidad del producto probablemente sea alta. Si el equipo no tiene experiencia y habilidades, un buen proceso delimita el daño, pero no conducirá, por sí mismo, a software de alta calidad.

Si los equipos son pequeños, es importante contar con una buena tecnología de desarrollo. El equipo no puede dedicar mucho tiempo a procedimientos administrativos tediosos. Los ingenieros invierten mucho de su tiempo diseñando y programando el sistema, por lo que las buenas herramientas afectan de forma importante a su productividad. Para proyectos grandes, un nivel básico de tecnología de desarrollo es esencial para la gestión de la información. Sin embargo, de forma paradójica, las sofisticadas herramientas CASE son en general menos importantes. Los miembros del equipo invierten relativamente poco tiempo en las actividades de desarrollo y más en comunicarse y comprender las otras partes del sistema. Éste es el factor dominante que afecta a su productividad. Las herramientas de desarrollo no son significativas en este caso.

La base del rectángulo de la Figura 28.3 es absolutamente crítica. La calidad del producto se ve afectada si un proyecto, independientemente de su tamaño, está mal presupuestado o planificado con un tiempo de entrega irreal. Un buen proceso requiere recursos

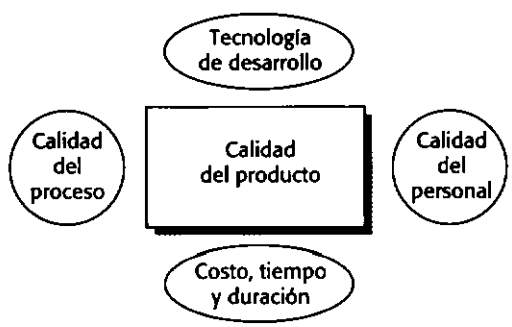


Figura 28.3 Principales factores de calidad del producto software.

para su implementación efectiva. Si estos recursos son insuficientes, el proceso no puede desarrollarse de forma efectiva. Si los recursos no son adecuados, sólo las personas excelentes pueden salvar el proyecto. Más aún, si el déficit es demasiado grande, la calidad del producto se degradará.

A menudo, la causa real de los problemas en la calidad del software no es la mala gestión, los procesos inadecuados o la poca calidad de la capacitación. Más bien, es el hecho de que las organizaciones deben competir para sobrevivir. Muchos proyectos de software infravaloran el esfuerzo o prometen una entrega rápida con el fin de conseguir el contrato de desarrollo. En un intento de mantener estos compromisos, la compañía podría sacrificar la calidad del software.

28.2 Clasificación de los procesos

Existen procesos de software en todas las organizaciones, desde empresas unipersonales hasta grandes multinacionales. Estos procesos son de diferentes tipos dependiendo del grado de formalización del proceso, los tipos de productos desarrollados y el tamaño de la organización, entre otros. Hay cuatro clases de procesos software:

1. *Procesos informales*. Son procesos en los que no existe un modelo de proceso definido de forma estricta. El proceso utilizado es elegido por el equipo de desarrollo. Los procesos informales podrían utilizar procedimientos formales, como la gestión de configuraciones, pero los procedimientos a utilizar y sus relaciones son definidos por el equipo de desarrollo.
2. *Procesos gestionados*. Se utiliza un modelo de proceso para dirigir el proceso de desarrollo. El modelo de proceso define los procedimientos, su agenda y las relaciones entre los procedimientos.
3. *Procesos metodológicos*. Se utiliza algún o algunos métodos de desarrollo definidos (como los métodos sistemáticos para diseño orientado a objetos). Estos procesos se benefician de la existencia de herramientas CASE para el diseño y el análisis.
4. *Procesos de mejora*. Son procesos que tienen inherentemente objetivos de mejora. Existe un presupuesto específico para estos procesos de mejora, y de procedimientos para introducir tales mejoras. Como parte de estas mejoras, se introducen mediciones cuantitativas del proceso.

Estas clasificaciones se solapan, por lo que un proceso puede ser de diferentes clases. Por ejemplo, el proceso puede ser informal en el sentido de que es seleccionado por el equipo de desarrollo. El equipo puede optar por usar un método de diseño particular. También tiene la capacidad de mejora de procesos. En este caso, el proceso se clasificaría como informal, metodológico y de mejora.

Esta clasificación es útil debido a que sirve como base para la mejora de procesos multidimensional. Ayuda a las organizaciones a elegir un proceso de desarrollo apropiado para los diferentes productos. La Figura 28.4 muestra los diferentes tipos de procesos que se podrían utilizar para el desarrollo de diferentes tipos de productos. En la figura no se muestran los procesos de mejora, ya que cualquier tipo de proceso puede ser un proceso de mejora.

Las clases de sistemas mostradas en la Figura 28.4 pueden solaparse. Por lo tanto, los sistemas pequeños a los que se les aplica reingeniería se pueden desarrollar utilizando un proceso metodológico. Los sistemas grandes siempre necesitan un proceso gestionado. Sin embargo, los mismos métodos de diseño no son recomendables para todo tipo de aplicaciones,

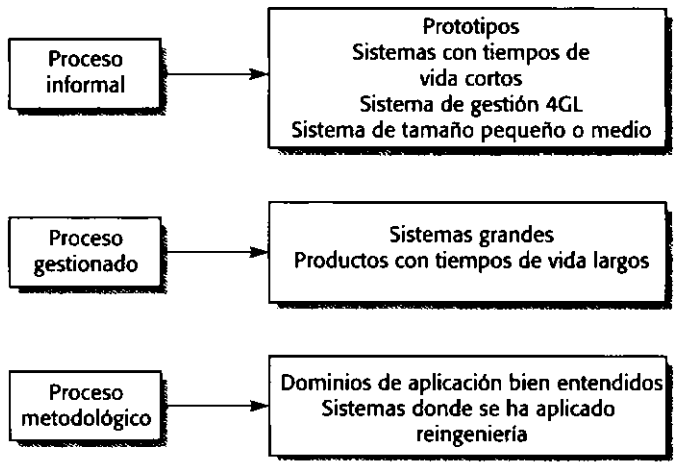


Figura 28.4 Aplicación de los procesos.

y los sistemas grandes constan de subsistemas de diferentes tipos. Por lo tanto, los sistemas grandes se desarrollan utilizando un proceso gestionado que no se basa en métodos de diseño particulares.

La clasificación de procesos proporciona una base para seleccionar el proceso correcto a utilizar cuando se desarrolla un tipo de producto particular. Por ejemplo, supongamos que se requiere un programa para ayudar a la migración de un tipo de computadora a otra. El tiempo de vida de este programa será relativamente corto. Su desarrollo no requiere los estándares, ni los procedimientos de gestión, que serían apropiados en el caso de que el sistema fuese utilizado durante muchos años.

La clasificación de procesos reconoce que el proceso afecta a la calidad del producto. Sin embargo, no supone que el proceso sea siempre el factor dominante. Provee una base para mejorar diversos tipos de procesos. Se aplican diferentes tipos de mejora de procesos a diferentes tipos de procesos. Por ejemplo, las mejoras para los procesos metodológicos se basan en mejores métodos de capacitación, mejor integración de los requerimientos y el diseño, herramientas CASE mejoradas, etc.

La mayoría de los procesos software tienen ahora el apoyo de herramientas CASE, por lo que son procesos con soporte. En la actualidad, los procesos metodológicos por lo general son apoyados por bancos de trabajo de análisis y diseño. Sin embargo, los procesos pueden tener otras clases de herramientas de apoyo (por ejemplo, herramientas para la creación de prototipos, herramientas de pruebas) independientemente de que se utilice o no un método de diseño estructurado.

El apoyo de las herramientas que puede ser efectivo en el apoyo de los procesos depende de la clasificación del proceso. Por ejemplo, los procesos informales pueden utilizar herramientas genéricas, como los lenguajes para la construcción de prototipos, compiladores, depuradores, procesadores de texto, etc. Sin embargo, raramente utilizan herramientas especializadas de una forma consistente. La Figura 28.5 muestra diversas herramientas a utilizar en el desarrollo de software. La efectividad de las herramientas depende del tipo de procesos utilizados.

Las herramientas CASE para análisis y diseño son las más efectivas cuando se sigue un proceso metodológico. Las herramientas especializadas dan soporte a actividades individuales. Por ejemplo, el equipo implicado en el desarrollo llevado a cabo desde diferentes localizaciones, puede crear una herramienta para mostrar cómo va el trabajo en cada sitio. A veces, estas herramientas especializadas deben ser desarrolladas especialmente para mejorar el proceso.

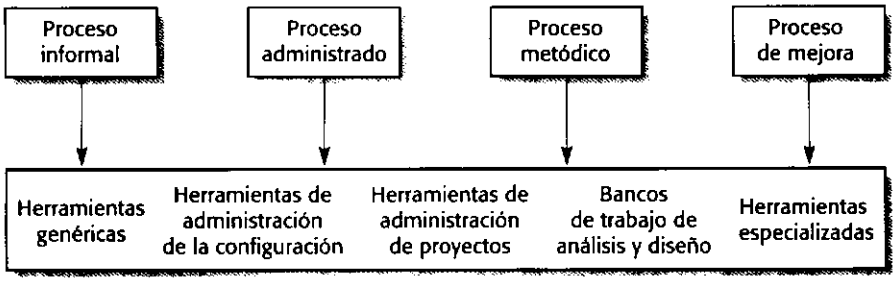


Figura 28.5 Herramientas de apoyo al proceso.

28.3 Medición del proceso

Las mediciones del proceso son datos cuantitativos de los procesos del software. Humphrey (Humphrey, 1989), en su libro sobre mejora de procesos, señala que la medición de los atributos de proceso y de producto es esencial para la mejora de procesos. También señala que las mediciones desempeñan un papel importante en la mejora de procesos de personal a pequeña escala (Humphrey, 1995). Las métricas de proceso se utilizan para evaluar si la eficiencia de un proceso ha mejorado. Por ejemplo, se puede medir el esfuerzo y tiempo dedicados a las pruebas. Las mejoras efectivas para los procesos de prueba reducen el esfuerzo, el tiempo de prueba o ambos. Sin embargo, las mediciones de procesos, por sí mismas, no se pueden utilizar para determinar si la calidad del producto ha mejorado. Las métricas de producto (véase el Capítulo 27) también deben hacerse y relacionarse con las actividades del proceso.

Se pueden recoger tres clases de métricas de proceso:

1. *El tiempo requerido para completar un proceso en particular.* Es el tiempo total dedicado al proceso, el tiempo de calendario, el tiempo invertido en el proceso por ingenieros particulares, etcétera.
2. *Los recursos requeridos para un proceso en particular.* Los recursos pueden ser el esfuerzo total en personas/día, los costes de viajes, los recursos de cómputo, etcétera.
3. *El número de ocurrencias de un evento en particular.* Ejemplos de eventos que se pueden supervisar son: el número de defectos descubiertos durante la inspección del código, el número de peticiones de cambios en los requerimientos, el número promedio de líneas de código modificadas en respuesta a un cambio de requerimientos, etcétera.

Los dos primeros tipos de mediciones se utilizan para ayudar a descubrir si los cambios en el proceso mejoran la eficiencia de un proceso. Por ejemplo, supongamos que existen puntos fijos en un proceso de desarrollo de software como la aceptación de requerimientos, la terminación de un diseño arquitectónico, la terminación de la generación de datos de prueba, etcétera. Es posible medir el tiempo y esfuerzo requeridos para moverse de uno de estos puntos fijos a otro. Los valores medidos se utilizan para indicar áreas donde se puede mejorar el proceso. Después de introducir los cambios, las mediciones de los atributos del sistema muestran si los cambios en el proceso han sido beneficiosos para reducir el tiempo o esfuerzo requerido.

Las mediciones del número de eventos que ocurren tienen una influencia directa en la calidad del software. Por ejemplo, incrementar el número de defectos descubiertos al cambiar el proceso de inspección del programa probablemente se reflejará en una mejora de la calidad del producto. Sin embargo, esto tiene que confirmarse por mediciones posteriores del producto.

La dificultad fundamental en la medición del proceso es conocer qué medir. Basili y Rombach (Basili y Rombach, 1988) propusieron el paradigma denominado GQM (Goal-Question-Metric). Éste se utiliza para ayudar a decidir qué mediciones tomar y cómo utilizarlas. Este enfoque se basa en la identificación de:

1. *Metas*. Lo que la organización está tratando de lograr. Ejemplos de metas son mejorar la productividad de los programadores, reducir los tiempos de desarrollo del producto, incrementar la fiabilidad del producto, etcétera.
2. *Preguntas*. Son refinamientos de las metas en las que se identifican las áreas específicas de incertidumbre relacionadas con las metas. Normalmente, una meta tendrá varias preguntas asociadas que requieren respuesta. Ejemplos de preguntas relacionadas con las metas anteriores son:
   - ¿Cómo se puede incrementar el número de líneas de código depuradas?
   - ¿Cómo se puede reducir el tiempo necesario para finalizar los requerimientos del producto?
   - ¿Cómo se puede llevar a cabo de forma más efectiva las evaluaciones de la fiabilidad?
3. *Métricas*. Son mediciones que hay que recoger para ayudar a responder las preguntas y confirmar si las mejoras del proceso ayudaron a cumplir la meta deseada. En los ejemplos anteriores, las mediciones que se podrían hacer son medir la productividad de los programadores individuales en líneas de código y su nivel de experiencia, medir el número de comunicaciones formales entre el cliente y el proveedor para cada cambio de los requerimientos y medir el número de pruebas requeridas para provocar una caída en el producto.

La ventaja de este enfoque cuando se aplica a la mejora de procesos es que separa las cuestiones organizacionales (las metas) de las cuestiones específicas del proceso (las preguntas). Se centra en la recolección de datos y señala que estos datos se deben analizar de diferentes formas, dependiendo de la pregunta que se pretenda contestar. Basili y Green (Basili y Green, 1993) describen cómo se utilizó este enfoque en un programa a largo plazo de mejora de procesos basado en las mediciones.

En el método ami de mejora de procesos del software (Pulford *et al.*, 1996), el enfoque GQM se desarrolló y combinó con el modelo de madurez de la capacidad del SEI. Los desarrolladores del método ami proponen un enfoque de etapas para la mejora de procesos en la que las mediciones se incluyen después de que la organización haya introducido algún tipo de disciplina en sus procesos. Esto proporciona guías y consejos prácticos al implementar la mejora de procesos basada en mediciones.

28.4 Análisis y modelado de procesos

El análisis y modelado de procesos comprende estudiar los procesos existentes y desarrollar un modelo abstracto de estos procesos que capte sus características principales. Estos modelos ayudarán a comprender el proceso y a comunicarlo a otros. A lo largo del libro, se han utilizado fragmentos de modelos de procesos para analizar actividades específicas como la ingeniería de requerimientos, el diseño de software, etcétera. Como se expuso en la Figura 28.2, el proceso de análisis sigue al proceso de medición. Ésta es una simplificación porque, en realidad, están entrelazadas. Se debe llevar a cabo un análisis previo para saber lo que se quie-

re medir y, cuando se hagan las mediciones, forzosamente se desarrollará una mejor comprensión del proceso de medición.

El análisis de procesos estudia los procesos existentes para comprender las relaciones entre las diferentes partes del proceso. Las etapas iniciales del análisis de procesos son inevitablemente cualitativas: el analista simplemente trata de descubrir las características principales del modelo. Las etapas posteriores son más cuantitativas y se utilizan diversas métricas de proceso. Después del análisis, los procesos se describen utilizando un modelo de procesos (Huff, 1996).

El punto inicial del análisis de procesos debe ser cualquiera de los modelos de procesos «formales». Muchas organizaciones cuentan con un modelo formal impuesto por el cliente del software. Este estándar define las actividades críticas y el ciclo de vida de los productos a entregar que se deben producir.

Los modelos formales sirven como un punto de inicio útil para el análisis de procesos. Sin embargo, raramente incluyen suficiente detalle o reflejan las actividades reales del desarrollo de software. Los modelos de procesos formales son más abstractos y sólo definen las actividades y productos a entregar del proceso principal. Por lo general, es necesario mirar «hacia adentro» del modelo para descubrir los procesos reales. Más aún, los procesos reales que se siguen a menudo difieren significativamente de los modelos formales, aunque por lo general se tengan que desarrollar los productos a entregar.

Las técnicas de análisis de procesos comprenden:

1. *Cuestionarios y entrevistas.* A los ingenieros que trabajan en un proyecto se les pregunta sobre lo que sucede realmente. Las respuestas a un cuestionario formal se refinan durante las entrevistas personales con los involucrados en el proceso.
2. *Estudios etnográficos.* Como se comentó en el Capítulo 7, los estudios etnográficos se utilizan para comprender la naturaleza del desarrollo de software como una actividad humana. Tal análisis revela la sutileza y complejidades no descubiertas por otras técnicas.

Cada uno de estos enfoques tiene ventajas y desventajas. El análisis basado en cuestionarios se lleva a cabo rápidamente una vez que se han diseñado las preguntas correctas. Sin embargo, si las preguntas no están bien redactadas o son inapropiadas, esto conducirá a un modelo incompleto o impreciso del proceso. Más aún, el análisis basado en cuestionarios es parecido a un formulario de evaluación. Los ingenieros encuestados dan las respuestas que creen que el encuestador desea escuchar en lugar de la verdad acerca del proceso utilizado.

Las entrevistas con la gente involucrada en el proceso son más flexibles que los cuestionarios. Se puede empezar con un guión de preguntas previamente preparado, pero adaptando éstas a las respuestas que se espera obtener de las diferentes personas. Si se da la oportunidad a los participantes de dialogar más ampliamente, se puede observar que éstos hablan acerca de los problemas del proceso, los cambios que está experimentando el proceso, etc.

El análisis etnográfico es más apropiado para descubrir los procesos que realmente se utilizan. Sin embargo, es una actividad costosa y a largo plazo que puede durar por lo menos varios meses. Se basa en la observación externa del proceso. Un análisis completo debe continuar desde las primeras etapas del proyecto hasta la entrega y mantenimiento del producto. Probablemente esto no es práctico en proyectos largos que duran varios años. El análisis etnográfico, por lo tanto, es más útil cuando se requiere un conocimiento profundo de los fragmentos del proceso. Deben llevarse a cabo estudios a pequeña escala centrándose en los detalles de proceso.

Los modelos de proceso son vistas simplificadas de los procesos de software, donde se muestran las actividades y las salidas del proceso. Los modelos de proceso utilizados en este libro son modelos abstractos simplificados que presentan una visión genérica de los procesos en cuestión. En este nivel abstracto, estos procesos son los mismos en muchas organizaciones. Sin embargo, estos modelos genéricos tienen diferentes instancias, dependiendo del tipo de software a desarrollar y del entorno organizacional. Los modelos detallados de procesos por lo general no son transferibles de una organización a otra.

Los modelos de procesos genéricos son una base útil para analizar los procesos. Sin embargo, no incluyen suficiente información para el análisis y mejora de procesos. Ésta requiere información de las actividades, los productos a entregar, las personas, las comunicaciones, la duración y otros procesos organizacionales que afectan al proceso de desarrollo. La Figura 28.6 explica lo que se podría incluir en un modelo de procesos detallado.

En un modelo de procesos también se registra el tiempo y las dependencias entre las actividades, los productos a entregar y las comunicaciones. A veces las actividades se llevan a cabo en paralelo y a veces en secuencia. Están entrelazadas de tal forma que el mismo ingeniero está involucrado en varias actividades. Los productos a entregar dependen de otros productos o de algunas comunicaciones entre los ingenieros que trabajan en el proceso.

En los ejemplos de los modelos de procesos del libro, se muestra la secuencia aproximada de actividades de izquierda a derecha. Las actividades que se llevan a cabo en paralelo se dibujan verticalmente, en la medida de lo posible.

Actividad (representada por un rectángulo redondeado sin sombra)	**Una actividad tiene claramente definidos un objetivo, las entradas y las condiciones de salida. Ejemplos de actividades son preparar un conjunto de datos de prueba para probar un módulo, codificar una función o módulo, hacer la prueba de lectura a un documento, etcétera. Por lo general, una actividad es atómica, es decir, incumbe a una persona o a un grupo. No se descompone en subactividades.**
Proceso (representado por un rectángulo redondeado con sombra)	**Un proceso es un conjunto de actividades que tiene alguna coherencia , cuyo objetivo, por lo general, se acuerda dentro de la organización. Ejemplos de procesos son el análisis de requerimientos, el diseño arquitectónico, la planificación de pruebas, etcétera.**
Producto a entregar (representado por un rectángulo con sombra)	**Un producto a entregar es una salida tangible de una actividad prevista en un plan de proyecto.**
Condición (representada por un paralelogramo)	**Una condición es o una precondición que debe cumplirse antes de iniciar un proceso o actividad o una postcondición que se cumple después de terminar el proceso o la actividad.**
Rol (representado por un círculo con sombra)	**Un rol es un área limitada de responsabilidad. Ejemplo de roles son el gestor de configuraciones, el ingeniero de pruebas, el diseñador de software, etc. Una persona puede tener varios roles distintos, y un solo rol se puede asociar a varias personas.**
Excepción (no se muestra en los ejemplos pero se representa como un cuadro de doble borde)	**Una excepción es una descripción de cómo modificar el proceso si ocurre un evento anticipado o no anticipado. Las excepciones a menudo son indefinidas y se deja a la habilidad de los administradores e ingenieros del proyecto el manejo de la excepción.**
Comunicación (representada por una flecha)	**Un intercambio de información entre personas o entre personas y sistemas de cómputo de apoyo. Las comunicaciones pueden ser informales o formales. Las comunicaciones formales podrían ser que un administrador del proyecto apruebe un producto a entregar; las comunicaciones informales podrían ser el intercambio de un correo electrónico para resolver las ambigüedades en un documento.**

Figura 28.6 Elementos de un modelo de proceso.

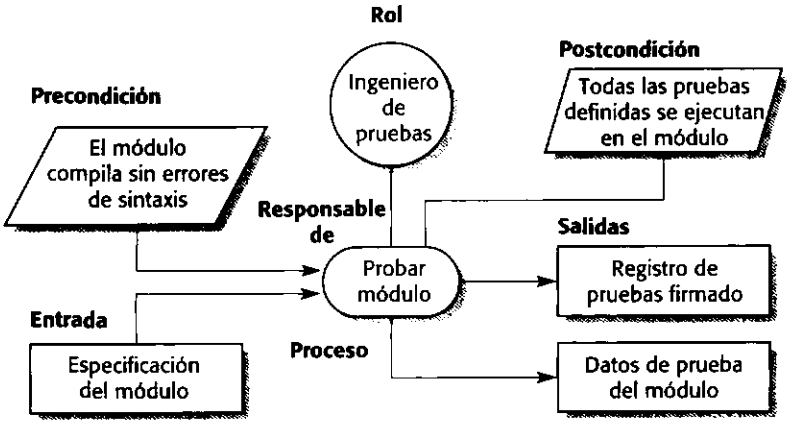


Figura 28.7 El proceso de pruebas de un módulo.

Los modelos de procesos detallados son sumamente complejos. Por ejemplo, consideremos los fragmentos de proceso mostrados en las Figuras 28.7 y 28.8. Éstos describen el proceso de probar un módulo de un sistema grande que utiliza un proceso de gestión de configuraciones controlado de forma estricta (véase el Capítulo 29). El software a probar y los datos de pruebas están bajo el control de la configuración. La Figura 28.7 muestra el rol responsable del proceso de prueba, las entradas y salidas del proceso y las pre y postcondiciones.

La Figura 28.8 descompone el proceso «Probar módulo» en varias actividades separadas. Este fragmento muestra sólo las actividades relativamente simples de la prueba del módulo. Existen cuatro series de actividades: preparar los datos de prueba, redactar un código de pruebas para un módulo, ejecutar las pruebas y hacer informes de pruebas. Las actividades en la

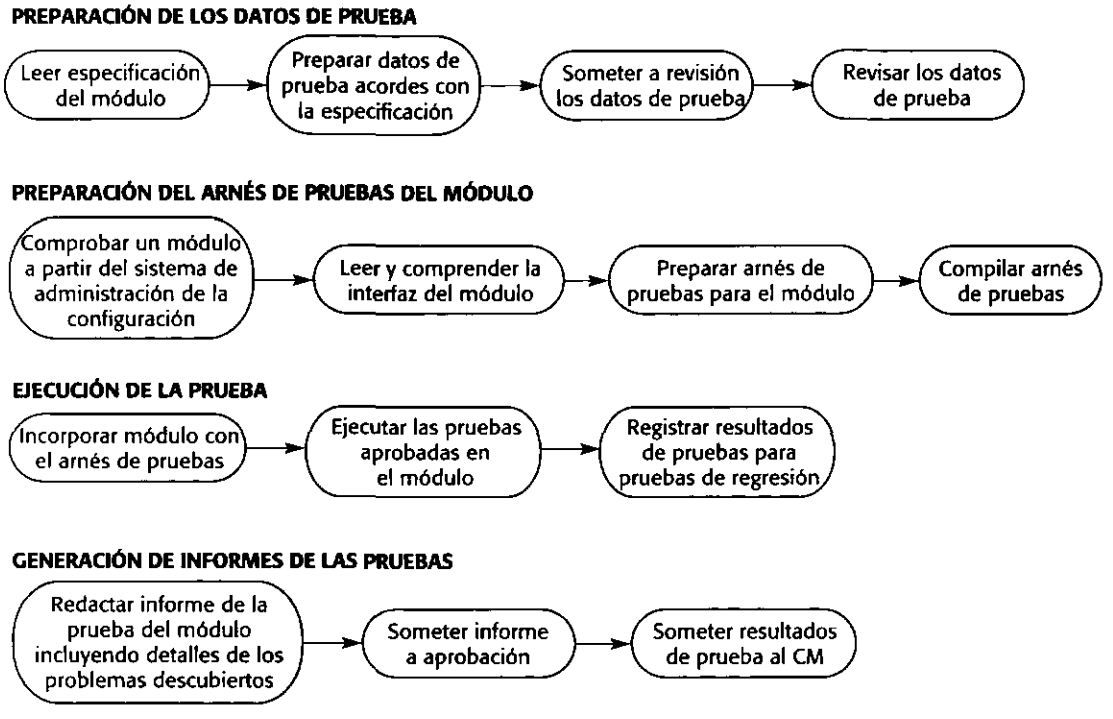


Figura 28.8 Las actividades involucradas en la prueba de un módulo.

serie de preparación normalmente están entrelazadas. Obviamente, las actividades de preparación preceden a la ejecución y elaboración del informe de actividades.

En este diagrama se ha dejado información sobre las pre y postcondiciones del proceso y sobre las entradas y salidas del proceso. Esta información haría que el modelo fuera complejo y difícil de comprender. En lugar de tratar de tener toda la información en un solo modelo, se necesita hacer varios modelos en diferentes niveles de abstracción. Éstos se relacionan por medio de los elementos comunes como las actividades y los productos a entregar. Algunos modelos se refieren primordialmente a las actividades del proceso; otros, a la información de control que dirige la ejecución del proceso.

28.4.1 Excepciones del proceso

Los procesos del software son entidades muy complejas. Aunque exista un modelo de procesos definido en una organización, éste sólo representa la situación ideal en la que el equipo de desarrollo se enfrenta con problemas no previstos. En la realidad, los problemas no previstos son un hecho de la vida diaria para los gestores del proyecto. El modelo de procesos «ideal» debe modificarse dinámicamente puesto que se deben encontrar soluciones a esos problemas. Ejemplos de estos tipos de excepciones a las que se enfrenta un administrador de proyectos son:

1. Varias personas clave enferman al mismo tiempo, justo antes de revisar un proyecto.
2. Una avería en la computadora de seguridad que deja todas las comunicaciones fuera de servicio durante varias horas.
3. Se lleva a cabo una reorganización de la organización, lo que significa que los gestores tienen que invertir gran parte de su tiempo trabajando en cuestiones organizacionales en lugar de la gestión del proyecto.
4. Se hace una petición no prevista de propuestas para nuevos proyectos. El esfuerzo se transfiere de un proyecto a trabajar en una propuesta.

En general, el efecto de un imprevisto es que, de alguna forma, los recursos, los presupuestos o los tiempos de un proyecto cambian. Es difícil predecir todos los imprevistos e incorporarlos en un modelo de proceso formal. Se tendrán que manejar dinámicamente estos imprevistos cambiando el proceso «estándar» para enfrentarse a estas circunstancias inesperadas. Por lo tanto, los modelos de proceso son inevitablemente incompletos, y el gestor del proyecto es responsable de tratar estos imprevistos y de adaptar el proceso como se requiera.

28.5 Cambio en los procesos

El cambio en el proceso significa hacer modificaciones en el proceso existente. Se puede hacer introduciendo nuevas prácticas, métodos o herramientas, cambiando el orden de las actividades, introduciendo o eliminando entregas del proceso, o introduciendo nuevos roles y responsabilidades. Deben fijarse metas para la mejora del proceso, como «reducir en un 25% el número de defectos descubiertos durante la prueba de integración». Estas metas deben conducir el cambio en el proceso y, una vez realizados cambios, deben utilizarse para evaluar el progreso.

Existen cinco etapas clave en el proceso de «cambio de procesos» (Figura 28.9):

1. *Identificación de la mejora.* Esta etapa comprende utilizar los resultados del análisis del proceso para identificar la calidad, la duración o los cuellos de botella de los cos-

tes donde los factores del proceso influyen de forma adversa en la calidad de producto. La mejora de procesos se centra en desvelar estos cuellos de botella proponiendo nuevos procedimientos, métodos y herramientas para abordar los problemas.

2. *Priorización de la mejora*. Esta etapa se centra en la evaluación de los cambios y en establecimiento de las prioridades para su implementación. Cuando se identifican muchos cambios posibles, normalmente es imposible introducirlos todos al mismo tiempo. Se tendrá que decidir cuáles son los más importantes. Pueden basarse las decisiones en las necesidades de mejora en áreas específicas del proceso, los costes de introducir los cambios, el impacto del cambio en la organización entre otros factores.
3. *Introducción del cambio del proceso*. La introducción del cambio del proceso significa agregar nuevos procedimientos, métodos y herramientas e integrarlas con otras actividades del proceso. Es importante dejar suficiente tiempo para introducir los cambios y asegurar que éstos son compatibles con otras actividades del proceso y con los procedimientos y estándares organizacionales.
4. *Capacitación en el cambio del proceso*. Sin capacitación, no es posible obtener los beneficios totales de los cambios del proceso. Pueden ser rechazados por los gestores y los ingenieros responsables de los proyectos de desarrollo. De igual forma, imponer los cambios del proceso sin la capacitación adecuada conduce a que estos cambios sean para degradar en lugar de mejorar la calidad del producto.
5. *Refinamiento del cambio*. Los cambios del proceso propuestos no son efectivos completamente al momento de introducirlos. Es necesario que exista una fase de ajuste donde se descubren problemas menores y se proponen e introducen las modificaciones al proceso. Esta fase de refinamiento dura varios meses hasta que los ingenieros de desarrollo están contentos con el nuevo proceso.

Una vez que se introduce un cambio, el proceso de mejora se itera con análisis adicionales para identificar problemas en el proceso, proponer mejoras, etcétera. No es práctico introducir muchos cambios al mismo tiempo. Aparte de los problemas de capacitación que esto provoca, introducir muchos cambios hace imposible evaluar el efecto de cada cambio en el proceso.

El gestor debe ser sensible a los sentimientos de la gente de su equipo cuando introduce cambios en el proceso. El proceso de reingeniería en los negocios (Hammer, 1990; Ould, 1995), de moda en los años 90, conllevó cambios radicales en los procesos, pero no fue satisfactorio en muchos casos debido a que no se tuvo en cuenta a las personas involucradas. Éstas sintieron que su experiencia fue soslayada y que sus puestos de trabajo cambiaron sin ningún tipo de consulta. Se resistieron a los cambios y aseguraron que con estos cambios no trabajarían.

No hay duda de que algunas personas se sienten amenazadas por el cambio o preocupadas por perder su trabajo o ser incapaces de adaptarse a las nuevas formas de trabajar. El gestor tiene que involucrar a todo el equipo en el proceso de cambio, entendiendo sus dudas e implicándolos en la planificación del proceso nuevo. Interesándolos en el proceso de cambio, será más fácil que ellos deseen hacerlo.

28.6 El marco de trabajo para la mejora de procesos CMMI

El software Engineering Institute (SEI) se estableció para mejorar las capacidades de la industria de software de los Estados Unidos de América. A mediados de los 80, el SEI inició

un estudio de las formas de evaluar las capacidades de los proveedores de software. El resultado de estos estudios fue el Modelo de Madurez de la Capacidad de Software del SEI (CMM) (Paulk *et al.*, 1993; Paulk *et al.*, 1995). Ha influido tremendamente en el convencimiento de la comunidad de ingeniería del software, para considerar seriamente la mejora de procesos. Al modelo CMM de software lo siguieron otros modelos de capacidad de madurez, entre ellos el Modelo de Madurez de la Capacidad de Personal (P-CMM) (Curtis *et al.*, 2001), expuesto en el Capítulo 25.

Otras organizaciones han desarrollado modelos de madurez del proceso similares. El modelo SPICE que aproxima a la valoración de la capacidad y a la mejora del proceso (Paulk y Konrad, 1994) es más flexible que el modelo del SEI. Incluye niveles de madurez similares a los niveles del SEI, pero además identifica procesos, como los procesos entre cliente y proveedor, que recorren estos niveles. A medida que subimos en nivel de madurez, el rendimiento de estos procesos clave debe mejorarse.

El proyecto Bootstrap tenía el objetivo de extender adaptar el modelo de madurez del SEI para hacerlo aplicable a un amplio espectro de compañías. El modelo Bootstrap (Haase *et al.*, 1994; Kuvaja *et al.*, 1994) utiliza los niveles de madurez del SEI, pero además incorpora:

1. Guías de calidad para ayudar en la mejora de procesos de las compañías.
2. Una distinción importante entre organización, metodología y tecnología.
3. Un modelo de proceso base (basado en el modelo utilizado por la Agencia Espacial Europea) que podría adoptarse.

En un intento de integrar la amalgama de modelos que se habían desarrollado (incluyendo sus propios modelos), el SEI se embarcó en un nuevo programa para desarrollar un modelo de capacidad integrado (CMMI). Éste sustituye al software y a los sistemas de ingeniería basados en CMM e integra a otros modelos de ingeniería. Tiene dos instancias, en etapas y continuo, y trata algunas de las debilidades del CMM de software.

El modelo CMMI (Ahern *et al.*, 2001) intenta ser un marco de trabajo para la mejora del proceso que sea aplicable en un amplio abanico de compañías. Su versión en etapas es compatible con el CMM de software y permite un desarrollo del sistema de la organización, la gestión de los procesos a valorar y su asignación a un nivel de madurez entre 1 y 5. Su versión continua permite una clasificación más detallada y considera 24 áreas de procesos (véase la Figura 28.10) en una escala de 1 a 6.

El modelo es muy complejo (su descripción tiene más de 1.000 páginas), por lo que hemos tenido que simplificarlo:

1. *Áreas de proceso.* El CMMI identifica 24 áreas de procesos que son relevantes para la capacidad y la mejora del proceso software. Éstas están organizadas en cuatro grupos en el modelo CMMI continuo. En la Figura 28.10 podemos ver estos grupos y sus correspondientes áreas.
2. *Metas.* Las metas son descripciones abstractas de un estado deseable que debería ser alcanzado por una organización. El CMMI tiene metas específicas asociadas a cada área de procesos y que definen el estado deseable para esta área. También tiene metas genéricas que son asociadas con la institucionalización de buenas prácticas. La Figura 28.11 muestra algunas metas específicas y genéricas en el CMMI.
3. *Prácticas.* Las prácticas en el CMMI son descripciones de vías para conseguir una meta. Se pueden asociar hasta siete prácticas específicas o genéricas con cada meta dentro de cada área de procesos. La Figura 28.12 muestra ejemplos de prácticas recomendadas. Sin embargo, el CMMI reconoce que lo importante es la meta, no el

Gestión del proceso	Definición de procesos organizacionales Centrar la atención en procesos organizacionales Aprendizaje organizacional Rendimiento de los procesos organizacionales Desarrollo e innovación organizacional
Gestión del proyecto	Planificación del proyecto Control y seguimiento del proyecto Gestión de acuerdos con los proveedores Gestión de la integración del proyecto Gestión de riesgos Integración del equipo Gestión cuantitativa del proyecto
Ingeniería	Gestión de requerimientos Desarrollo de requerimientos Soluciones técnicas Integración del producto Verificación Validación
Soporte	Gestión de configuraciones Gestión de calidad del proceso y del producto Análisis y mediciones Análisis y toma de decisiones Entorno organizacional para integración Análisis y resolución causal

Figura 28.10
Áreas de proceso en el CMMI.

camino que lleva a ella. Las organizaciones utilizan cualquier práctica apropiada para alcanzar cualquier meta del CMMI; no tienen por qué seguir las recomendaciones del CMMI.

Las metas y prácticas genéricas no son técnicas, pero están asociadas con la institucionalización de las buenas prácticas, lo que significa que dependen de la madurez de la organización. Por lo tanto, en una organización nueva que se halla en una etapa temprana del desarrollo de la madurez, la institucionalización puede significar el seguimiento de los planes y los procesos establecidos. Sin embargo, en una organización con más madurez, procesos avanzados, la institucionalización puede significar controlar los procesos utilizando técnicas estadísticas u otras técnicas cuantitativas.

Gestión de acciones correctivas cuando el rendimiento del proyecto o sus resultados se desvían significativamente del plan	Meta específica en el seguimiento y control del proyecto
El rendimiento actual y el progreso del proyecto es monitorizado comparándolo con el plan del proyecto	Meta específica en el seguimiento y control del proyecto
Los requerimientos son analizados y validados, se desarrolla una definición de las funcionalidades requeridas	Meta específica en desarrollo de requerimientos
Se determinan las principales causas de defectos y otros problemas	Meta específica en análisis y resolución causal
El proceso es institucionalizado como un proceso definido	Meta genérica

Figura 28.11
Ejemplos de metas en el CMMI.

Análisis de los requerimientos derivados para asegurar que son necesarios y suficientes **Validar los requerimientos para asegurar que el producto resultante funciona tal y como se pretendía en el entorno del usuario, utilizando múltiples técnicas cuando sea apropiado.**	**Los requerimientos son analizados y validados, y se define la funcionalidad requerida.**
Seleccionar los defectos y otros problemas a analizar **Análisis causal de los defectos y otros problemas seleccionados y proponer acciones para corregirlos**	**Se determinan sistemáticamente las principales causas de defectos y otros problemas**
Establecer y mantener una política organizacional de planificación y presentación del proceso de desarrollo de requerimientos **Asignar responsabilidades y autorizaciones para presentar el proceso, desarrollar los productos de trabajo y proveer los servicios para el proceso de desarrollo de requerimientos**	**El proceso es institucionalizado como un proceso definido**

Figura 28.12 Prácticas y metas asociadas en el CMMI.

La valoración de un CMMI implica examinar los procesos en una organización y clasificarlos en una escala de seis puntos que reflejan el nivel de madurez en cada área de proceso. A un proceso se le asigna el nivel en la escala de seis puntos como sigue:

1. *No productivo.* No se satisfacen una o más de las metas específicas asociadas con el área de proceso.
2. *Productivo.* Se satisfacen las metas asociadas al área de proceso, y para todos los procesos el ámbito del trabajo a realizar es fijado y comunicado a los miembros del equipo.
3. *Gestionado.* A este nivel, las metas asociadas con el área de proceso son conocidas y tienen lugar políticas organizacionales que definen cuándo se debe utilizar cada proceso. Debe haber planes documentados, gestión de recursos y monitorización de procedimientos a través de la institución.
4. *Definido.* Este nivel se centra en la estandarización organizacional y el desarrollo de procesos. Cada proyecto de la organización tiene un proceso de gestión creado a medida desde un conjunto de procesos organizacionales. La información y las medidas del proceso son recogidas y utilizadas para las mejoras futuras del proceso.
5. *Gestionado cuantitativamente.* En este nivel, existe una responsabilidad organizacional de usar métodos estadísticos y otros métodos cuantitativos para controlar los subprocesos. Esto significa que en el proceso de gestión debemos utilizar medidas del proceso y del producto.
6 *Optimizado.* En este nivel superior, la organización debe utilizar medidas de proceso y de producto para dirigir el proceso de mejora. Debemos analizar las tendencias y adaptar los procesos a las necesidades de los cambios del negocio.

Por supuesto, esto es una descripción simplificada de los niveles de capacidad, pero se puede pensar en estos niveles como algo progresivo, con una descripción explícita de los procesos en los niveles más bajos, y donde las mediciones de proceso y producto dirigirán la estandarización del proceso para cambiarlo y mejorarlo.

28.6.1 El modelo CMMI en etapas

El modelo CMMI de niveles es comparable con el modelo CMM de software en el sentido en que provee una forma de valorar la capacidad del proceso de una organización clasificándola en uno de cinco niveles. El modelo describe las metas que se deben alcanzar en cada uno de estos niveles. La mejora de procesos se lleva a cabo implementando prácticas en cada nivel, subiendo desde el nivel inferior hasta el superior del modelo. La Figura 28.13 muestra los cinco niveles de este modelo CMMI.

Cada nivel de madurez tiene asociado un conjunto de áreas de proceso y metas genéricas. Por ejemplo, las áreas de proceso definidas para el segundo nivel del modelo (nivel gestionado) son:

1. *Gestión de requerimientos.* Gestiona los requerimientos del proyecto y de sus componentes, e identifica inconsistencias entre estos requerimientos y el plan y los productos del proyecto.
2. *Planificación del proyecto.* Establece y mantiene los planes, los cuales definen las actividades del proyecto.
3. *Seguimiento y control del proyecto.* Provee la comprensión del progreso del proyecto y la aplicación de medidas correctivas cuando el rendimiento del proyecto se desvía significativamente del plan.
4. *Acuerdos con los proveedores.* Gestiona la adquisición de productos y servicios de proveedores externos al proyecto con los cuales existen acuerdos formales.
5. *Análisis y mediciones.* Desarrolla y mantiene una capacidad de medición que se utiliza en el soporte de gestión de la información.
6. *Garantía de la calidad del proceso y del producto.* Provee personal y gestión con comprensión objetiva del proceso y los productos de trabajo asociados.
7. *Gestión de configuraciones.* Establece y mantiene la integridad de los productos de trabajo usando identificación, control y estado de configuraciones, así como auditorías.

Así como estas prácticas específicas, la operativa de las organizaciones en el segundo nivel en el modelo CMMI debe haber alcanzado metas genéricas de institucionalización donde cada uno de los procesos sea un proceso gestionado. Ejemplos de prácticas institu-

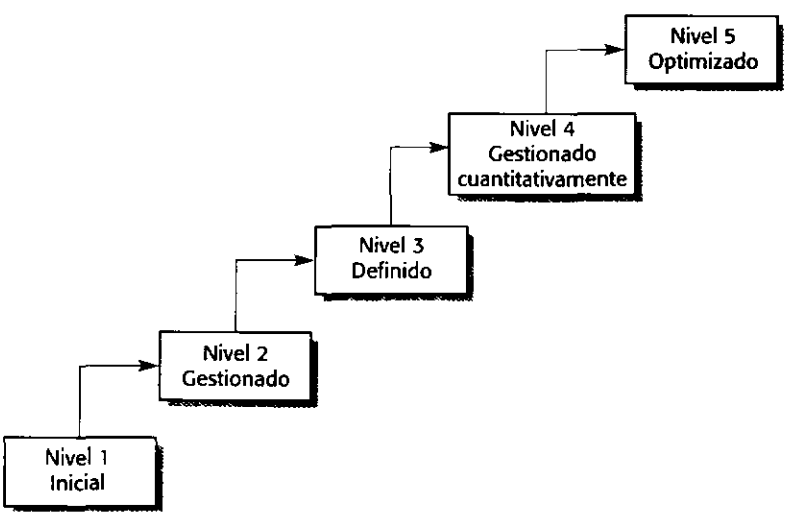


Figura 28.13
El modelo CMMI de niveles.

cionales asociadas con la planificación del proyecto que permiten a este proceso ser un proceso gestionado son:

- Establecer y mantener una política organizacional de planificación y mejora del proceso de planificación.
- Proveer los recursos adecuados para mejorar el proceso de gestión del proyecto, desarrollando las herramientas y proveyendo los servicios al proceso.
- Seguimiento y control del proyecto y aplicación de las medidas correctivas adecuadas cuando sea necesario.
- Revisión de actividades, estado y resultados del proceso de planificación del proyecto con una gestión a alto nivel y resolución de problemas.

La ventaja del modelo CMMI en etapas, aparte de su compatibilidad con el modelo CMM, es que define un camino claro para la mejora de las organizaciones. Éstas subirán del segundo al tercer nivel, y así sucesivamente. La desventaja es que podría ser más adecuado introducir metas y prácticas correspondientes a los niveles superiores antes que las prácticas de niveles inferiores. Cuando una organización hace esto, la valoración de su madurez da una imagen engañosa.

28.6.2 El modelo CMMI continuo

Los modelos de madurez continuos no clasifican a las organizaciones en niveles discretos. Éstos son modelos que hilan más fino y que consideran prácticas individuales, grupos de éstas y sus valoraciones. La valoración de la madurez no es, por lo tanto, un solo valor, sino un conjunto de valores que muestran la madurez de la organización para cada proceso o grupo de procesos.

El modelo CMMI continuo evalúa cada área de proceso (véase la Figura 28.10) y le asigna una nivel de valoración entre 1 y 6 (como ya describimos) a cada área de proceso.

Normalmente, las organizaciones operan a diferentes niveles de madurez en distintas áreas de proceso. En consecuencia, el resultado de una valoración con el modelo CMMI continuo es un perfil de capacidad que muestra cada área de proceso y su correspondiente valoración de capacidad. La Figura 28.14 muestra un fragmento de un perfil de capacidad donde pode-

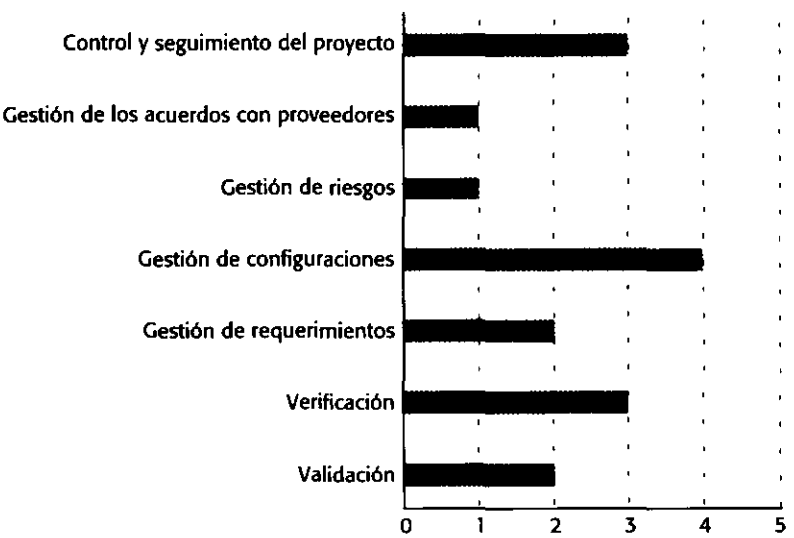


Figura 28.14
Un perfil de capacidad de proceso.

mos ver diferentes procesos con diferentes niveles de capacidad. Las organizaciones pueden desarrollar perfiles de capacidad actuales o futuros, donde estos perfiles futuros reflejarían el nivel de capacidad en cada área de proceso que se espera alcanzar.

La principal ventaja del modelo continuo es que las organizaciones pueden elegir procesos de mejora de acuerdo con sus propias necesidades y requerimientos. La experiencia demuestra que diferentes tipos de organizaciones tienen distintos requerimientos en su mejora de procesos. Por ejemplo, una empresa que desarrolla software para la industria aeroespacial puede centrarse en mejoras de la especificación del sistema, gestión de configuraciones y validación, mientras que una empresa de desarrollo Web estaría más interesa en los procesos cara al cliente. El modelo de niveles requiere que las compañías se centren en los diferentes niveles sucesivamente. Sin embargo, el modelo CMMI continuo permite más flexibilidad manteniendo la ayuda del CMMI.

PUNTOS CLAVE

- **La mejora de procesos comprende el anális del proceso, la estandarización, la medición y el cambio. El aprendizaje es esencial si la mejora de proceso va a llevarse a cabo.**
- **Los procesos pueden clasificarse como informales, gestionados, metodológicos y de mejora. Podemos utilizar esta clasificación para identificar las herramientas de soporte al proceso.**
- **El ciclo de mejora del proceso comprende medición, análisis del proceso y cambio del modelado y del proceso.**
- **La medición debe ser usada para responder a preguntas específicas del proceso software utilizado. Estas cuestiones deben basarse en metas de mejora organizacionales.**
- **Se pueden usar tres métricas de proceso: métricas de tiempo, de utilización de recursos y de eventos.**
- **Los modelos de proceso incluyen descripciones de actividades, subprocesos, roles, excepciones, comunicaciones, entregas y otros procesos.**
- **El modelo de madurez de proceso CMMI es un modelo de mejora de procesos integrado que soporta la mejora de procesos en etapas y continuo.**
- **En el modelo CMMI, la mejora de procesos está basada en alcanzar un conjunto de metas relacionadas con las buenas prácticas de ingeniería del software y describir, analizar y controlar las prácticas utilizadas para alcanzar estas metas. El modelo CMMI incluye prácticas recomendadas que pueden utilizarse, pero no obliga a utilizarlas.**

LECTURAS ADICIONALES

CMMI Distilled. En el momento de escritura de este libro, el único resumen conciso sobre el modelo CMMI. Es fácil de leer si ya se comprende el modelo CMM, ya que le falta una introducción general a la mejora de procesos. (D. M. Ahern *et al.*, 2001, Addison-Wesley.)

«Can you trust software capability evaluations». Este artículo hace una mirada escéptica al tema de la evaluación de la capacidad y expone por qué estas evaluaciones no dan una visión real de la madurez de la organización. [E. O'-Connell y Saiedian, *IEEE Computer*, 32(2), febrero de 2000.]

Trends in Software: Software Process Modelling and Technology. Este libro contiene una buena selección de artículos que tratan los diferentes aspectos de procesos software, incluyendo procesos de modelado, procesos de soporte y el uso del CMM. (A. Fuggetta y A. Wolf (eds.), 1996, John Wiley & Sons.)

Managing the Software Process. Un texto clásico sobre mejora de procesos software y el primero que publicó la noción de aproximación estructurada a la mejora de procesos. A pesar de que el libro ya tiene bastantes años, los consejos que contiene siguen siendo muy relevantes, especialmente en proyectos software grandes. (W. S. Humphrey, 1988, Addison-Wesley.)

EJERCICIOS

28.1 Sugiera modelos de proceso para las siguientes acciones:

- Encender una fogata de leña
- Cocinar una comida de tres platos (los cuales se eligen en un menú)
- Escribir un programa pequeño (50 líneas)

28.2 ¿En qué circunstancias se determina la calidad del producto a partir de la calidad del equipo de desarrollo? Dé ejemplos de los tipos de productos de software que dependen particularmente del talento y la habilidad individual.

28.3 Explique por qué un proceso metodológico no es necesariamente un proceso gestionado, como se define en la Sección 28.2.

28.4 Sugiera tres herramientas software especializadas que puedan desarrollarse para dar suporte al programa de mejora de procesos de una organización.

28.5 Suponga que la meta de la mejora de procesos en una organización es incrementar el número de componentes reutilizables que se producen durante el desarrollo. Sugiera tres cuestiones en el paradigma GQM que conduzcan a esto.

28.6 Describa tres tipos de métricas de proceso software que puedan ser parte de un proceso de mejora de procesos. Dé un ejemplo de cada tipo de métrica.

28.7 Diseñe un proceso para valorar y priorizar propuestas de cambio en el proceso. Documente este proceso como un modelo de proceso que muestra los roles que intervienen en él.

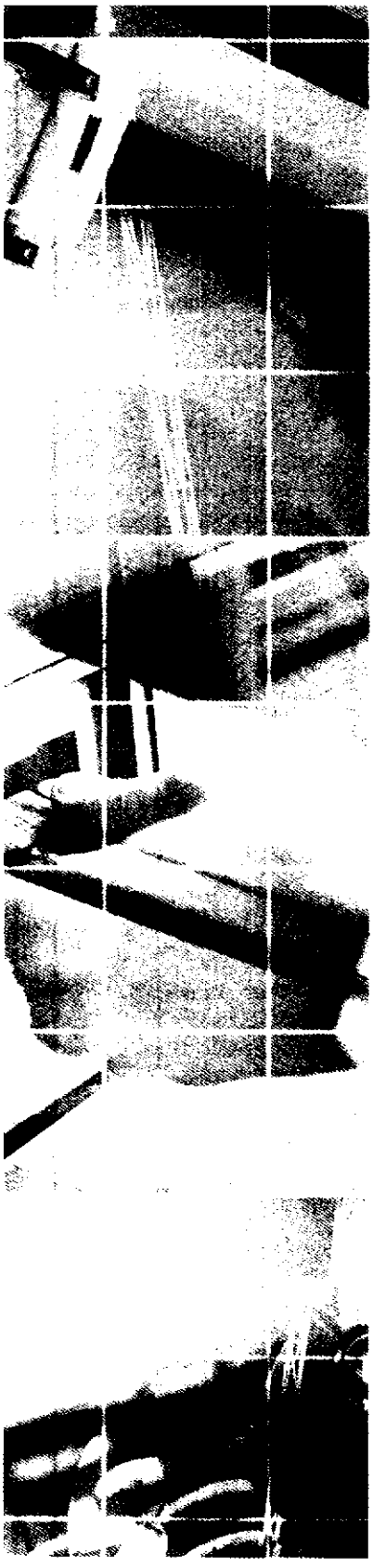
28.8 Señale dos ventajas y dos desventajas del enfoque de la valoración y la mejora del proceso incorporada en los marcos de trabajo de mejora de procesos como CMMI.

28.9 ¿En qué circunstancias debería recomendar el uso de la representación por niveles del CMMI?

28.10 ¿Qué diferencias hay entre metas genéricas y específicas en el CMMI?

28.11 ¿Cuáles son la ventajas y desventajas del uso modelos de madurez basados en metas frente a los basados en prácticas?

28.12 ¿Son intrínsecamente deshumanizados los programas de mejora de procesos que implican medir el trabajo de las personas en el proceso y que cambian dicho proceso? ¿Qué resistencia puede surgir en un programa de mejora de procesos?



29 Gestión de configuraciones

Objetivos

El objetivo de este capítulo es introducir el proceso de gestión del código y la documentación de un sistema de software evolutivo. Cuando termine de leer este capítulo:

- comprenderá por qué es importante la gestión de la configuración del software;
- habrá sido introducido en cuatro actividades principales de la gestión de configuraciones: planificación de la gestión de configuraciones, gestión de los cambios, gestión de versiones y entregas, y construcción de sistemas;
- aprenderá cómo se pueden utilizar las herramientas CASE para apoyar los procesos de gestión de configuraciones.

Contenidos

La gestión de configuraciones (CM) es el desarrollo y aplicación de estándares y procedimientos para gestionar un sistema software evolutivo. Como se expuso en el Capítulo 7, los requerimientos del sistema siempre cambian durante su desarrollo y su uso, y se tienen que incorporar estos requerimientos en nuevas versiones del sistema. Es necesario gestionar estos sistemas que evolucionan, porque es fácil perder la pista de los cambios que se han incorporado dentro de cada versión. Las versiones incorporan propuestas de cambios, correcciones de fallos y, adaptaciones para hardware y sistemas operativos diferentes. Pueden existir varias versiones en desarrollo y en uso al mismo tiempo. Si no se tienen unos procedimientos de gestión de configuraciones adecuados, se puede hacer un esfuerzo inútil modificando la versión errónea de un sistema, entregar una versión incorrecta a los clientes o perder la pista de dónde ha sido guardado el código fuente.

Los procedimientos de gestión de configuraciones definen cómo registrar y procesar los cambios propuestos al sistema, cómo relacionar éstos con los componentes del sistema y los métodos utilizados para identificar las diversas versiones del sistema. Las herramientas de gestión de configuraciones se utilizan para almacenar las versiones de los componentes del sistema, construir sistemas a partir de estos componentes y llevar el registro de entregas de las versiones del sistema a los clientes.

Algunas veces, la gestión de configuraciones es parte de un proceso general de gestión de la calidad del software (expuesto en el Capítulo 27), donde el mismo gestor comparte las responsabilidades de la gestión de la calidad y de la configuración. Los desarrolladores del software entregan éste al equipo de garantía de calidad, quienes son responsables de la comprobación de que el sistema es de calidad aceptable. Aquí comienza a ser un sistema controlado, lo que significa que los cambios en el sistema tienen que ser acordados y registrados antes de ser implementados. Algunas veces, los sistemas controlados se denominan «líneas base» puesto que son el punto de inicio para la evolución controlada.

Hay muchas razones por las que los sistemas existen en diversas configuraciones. Éstas se producen para diferentes computadoras, para diferentes sistemas operativos, para incorporar funciones específicas del cliente, etcétera (véase la Figura 29.1). Los gestores de la configuración son responsables de llevar los registros de las diferencias entre las versiones del software, para asegurar que las nuevas versiones se deriven de forma controlada y para entregar las nuevas versiones a los clientes correctos en el momento justo.

La definición y uso de gestión de configuraciones es esencial para la certificación de calidad ISO 9000 y para los estándares CMM y CMMI (Paulk *et al.*, 1995; Ahern *et al*, 2001; Peach, 1996). Un ejemplo de tal estándar es el IEEE 828-1998, que define un estándar para los planes de la gestión de configuraciones. Dentro de una organización, estos estándares se publican en un manual de gestión de configuraciones o como parte del manual de calidad. Los estándares externos se utilizan como base para estándares organizacionales detallados y se ajustan a un entorno específico.

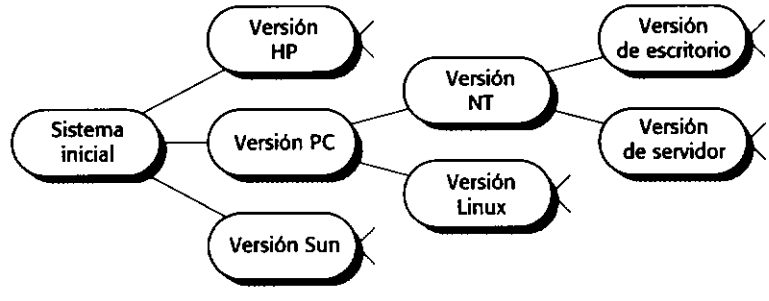


Figura 29.1
Familias de sistemas.

En un proceso tradicional de desarrollo de software basado en el modelo en «cascada» (véase el Capítulo 4), el software se entrega al equipo de gestión de configuraciones después de que el desarrollo haya sido completado y se hayan probado los componentes de software. Luego este equipo tiene la responsabilidad de construir el sistema completo y gestionar las pruebas del sistema. Los fallos encontrados durante las pruebas del sistema se devuelven al equipo de desarrollo para su reparación. A continuación reparan el fallo y entregan una nueva versión del componente reparado al equipo de la garantía de calidad. Si la calidad es aceptable, éste pasa a ser la nueva línea base para el desarrollo del sistema.

Este modelo, donde el equipo de garantía de calidad controla la integración del sistema y el proceso de pruebas, ha influenciado el desarrollo de estándares para la gestión de configuraciones. Muchos estándares de garantía de calidad suponen que se utiliza el modelo en cascada para el proceso del software en el desarrollo de sistemas (Bersoff y Davis, 1991). Esto significa que los estándares tienen que adaptarse a las aproximaciones modernas de desarrollo de software basadas en especificación y desarrollo incremental. Hass (Hass, 2003) discute alguna de estas adaptaciones para los procesos de desarrollo software tales como el desarrollo ágil.

Para ajustarse a este desarrollo incremental, algunas organizaciones han desarrollado un nuevo enfoque para gestionar las configuraciones que permite el desarrollo concurrente y las pruebas del sistema. Este enfoque se basa en la forma regular (a menudo diaria) de construcción del sistema completo a partir de sus componentes:

1. La organización que lleva a cabo el desarrollo fija una hora de entrega (por ejemplo, a las 14.00 horas) de los componentes del sistema. Si los desarrolladores tienen nuevas versiones de los componentes que están escribiendo, entonces deben entregarlos todos a esa hora. Los componentes podrían estar incompletos, pero deben ser capaces de proveer alguna funcionalidad básica que se pueda probar.
2. Una nueva versión del sistema se construye a partir de estos componentes compilándolos y vinculándolos para formar un sistema completo.
3. Entonces el sistema se entrega al equipo de pruebas, que lleva a cabo un conjunto predefinido de pruebas sobre el sistema. Al mismo tiempo, los desarrolladores siguen trabajando en sus componentes, agregando funcionalidades y reparando los fallos descubiertos en las pruebas previas.
4. Los fallos encontrados durante las pruebas del sistema se documentan y se devuelven a los desarrolladores del sistema. Éstos reparan dichos fallos en una versión ulterior del componente.

La principal ventaja de utilizar construcciones diarias del software es que se incrementan las oportunidades de encontrar problemas que surgen a partir de las interacciones al inicio del proceso. Más aún, las construcciones diarias provocan las pruebas de las unidades de los componentes. Psicológicamente, los desarrolladores se ven presionados «para no derribar el edificio», es decir, para no entregar versiones de los componentes que provoquen un fallo en todo el sistema. Por lo tanto, son reticentes a entregar nuevas versiones de los componentes que no se hayan probado adecuadamente. Si se encuentran fallos en el software y éstas se abordan durante las pruebas de las unidades, se invierte menos tiempo en las pruebas del sistema.

La utilización exitosa de las construcciones diarias requiere procesos de gestión del cambio muy rigurosos para llevar un registro de los problemas encontrados y resueltos. También conduce a la gestión de un número muy grande de versiones del sistema y de los componentes. Por lo tanto, una buena gestión de configuraciones es esencial para que este enfoque tenga éxito.

La gestión de configuraciones en el desarrollo ágil y desarrollo rápido no pueden basarse en rígidos procedimientos y papeleo burocrático. Aunque éstos pueden ser necesarios para proyectos grandes o complejos, pueden ralentizar el proceso de desarrollo. Mantener registros cuidadosos es esencial para sistemas grandes y complejos desarrollados desde diferentes ubicaciones, pero innecesario en proyectos pequeños. En estos proyectos, todos los miembros del equipo trabajan juntos en la misma habitación, y la sobrecarga asociada a mantener los registros ralentiza el proceso de desarrollo. Sin embargo, esto no significa que, cuando se requiera un desarrollo rápido, la gestión de configuraciones deba ser totalmente abandonada. Los procesos ágiles utilizan herramientas simples de gestión de configuraciones, como un gestor de versiones y herramientas para la construcción del sistema, que incorporarán algo de control. Todos los miembros del equipo tienen que aprender a utilizar estas herramientas y asumir las disciplinas que ellas imponen.

29.1 Planificación de la gestión de configuraciones

Un plan de gestión de configuraciones describe los estándares y procedimientos utilizados para la gestión de la configuración. El punto de inicio para desarrollar el plan es un conjunto de estándares generales de gestión de la configuración de toda la compañía adaptables a cada proyecto específico. El plan de la CM se organiza en varios capítulos que incluyen:

1. La definición de lo que se debe gestionar (los elementos de configuración) y el esquema formal para identificar estas entidades.
2. Un enunciado de quién toma la responsabilidad de los procedimientos de gestión de configuraciones y quién envía las entidades controladas al equipo de gestión de configuraciones.
3. Las políticas de gestión de configuraciones utilizadas para gestionar el control de los cambios y las versiones.
4. Una descripción de las herramientas a utilizar para la gestión de configuraciones y el proceso a aplicar cuando se utilizan estas herramientas.
5. Una definición de la base de datos de la configuración que se utilizará para registrar la información de la configuración.

En el plan de la CM se incluye información adicional de la gestión del software por parte de los proveedores externos y los procesos de auditoría para el proceso de la CM.

Una parte importante del plan de la CM es la definición de responsabilidades. Define quién es el responsable de la entrega de cada documento o de cada componente de software para la garantía de la calidad y la gestión de la configuraciones. También define los revisores de cada documento. La persona responsable de la entrega de los documentos no es preciso que sea la misma responsable de producir el documento. Para simplificar las interfaces, a menudo es conveniente hacer que los gestores de proyectos o los líderes del equipo sean responsables de todos los documentos producidos por su equipo.

29.1.1 Identificación de los elementos de configuración

En un sistema grande de software, puede haber cientos de módulos de código fuente, scripts de pruebas, documentos de diseño, etc. Éstos son producidos por diferentes personas y, cuando fueron creados, pudieron tener nombres similares. Para seguir el registro de toda esta in-

formación y encontrar el fichero adecuado cuando éste se precise, se necesitará un esquema consistente de identificación para todos los elementos del sistema de gestión de configuraciones.

Durante el proceso de planificación de la gestión de configuraciones, se decide exactamente qué elementos (o clases de elementos) se van a controlar. Los documentos o grupos de documentos relacionados del control de la configuración son documentos formales o elementos de la configuración. Normalmente, los planes del proyecto, las especificaciones, los diseños, los programas y los conjuntos de datos de prueba son elementos de la configuración. Todos los documentos que son necesarios para el mantenimiento futuro del sistema deben ser controlados por el sistema de control de configuraciones.

Sin embargo, esto no significa que todos los documentos o archivos deban estar bajo el control de configuraciones. Documentos como, por ejemplo, documentos técnicos de trabajo que presentan la captura de ideas para el posterior desarrollo, minutas de las reuniones de grupo, bosquejo del plan y propuestas, no tendrán relevancia a largo plazo y no serán necesarios para el futuro mantenimiento del sistema.

El esquema de asignación de nombres a los documentos debe asignar un nombre único a todos los documentos de control de la configuración. Este nombre debe reflejar el tipo de elemento, la parte del sistema en la que se utiliza y el creador del elemento, entre otros. En su esquema de nombres, también deberá reflejar las relaciones entre elementos para asegurar que los documentos relacionados tengan una raíz común de su nombre. Esto conduce a un esquema de asignación de nombres jerárquico. Algunos ejemplos de los nombres son:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE
PCL-TOOLS/EDIT/HELP/QUERY/HELPFRAMES/FR-1

La parte inicial del nombre es el nombre del proyecto, PCL-TOOLS. En este proyecto, existen cuatro herramientas diferentes; el nombre de la herramienta (EDIT) se utiliza como la siguiente parte del nombre. Cada herramienta está compuesta de módulos nombrados de forma distinta, cuyos nombres pasan a ser parte del identificador (FORMS, HELP). Este proceso de descomposición continúa hasta que se haga referencia a los documentos formales en el nivel base (véase la Figura 29.2). Las hojas de la jerarquía de la documentación son elementos de configuración formales. La Figura 29.2 muestra que se requieren tres documentos

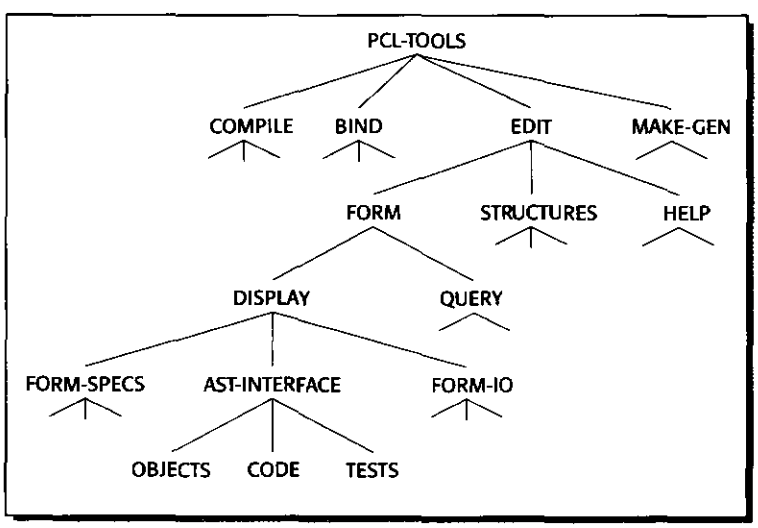


Figura 29.2 Jerarquía de la configuración usada para la asignación de identificadores.

formales para cada componente: una descripción de los objetos (OBJECTS), el código del componente (CODE) y un conjunto de pruebas para el código (TEST). Elementos como la ayuda son también gestionados y tienen diferentes nombres (FR-1, en el ejemplo anterior).

La asignación de nombres jerárquica es simple y fácil de entender, y a veces copia la estructura de directorios utilizada para almacenar los archivos del proyecto. Sin embargo, reflejan la estructura del proyecto cuando se desarrolló el software. Los nombres de los elementos de configuración asociados a un proyecto particular pueden reducir las oportunidades de reutilización. Puede ser muy difícil encontrar componentes relacionados (por ejemplo, todos los componentes desarrollados por el mismo programador) donde el esquema de nombres no refleja esta relación.

29.1.2 La base de datos de configuraciones

La base de datos de configuraciones se utiliza para registrar toda la información relacionada con las configuraciones y sus elementos. Sus funciones principales son ayudar a la evaluación del impacto de los cambios en el sistema y proveer información de la gestión acerca del proceso de la CM. Además de definir el esquema de la base de datos de la configuración, como parte del proceso de planificación de la CM también se deben definir los formularios y los procedimientos para registrar y recuperar la información del proyecto.

Una base de datos de configuraciones no incluye información acerca de los elementos de configuración. Registra información acerca de los usuarios de los componentes, los clientes del sistema, plataformas de ejecución, cambios propuestos, etc. Le debe ser posible suministrar respuestas a una variedad de consultas acerca de las configuraciones del sistema. Algunas consultas podrían ser:

1. ¿A qué clientes se les ha entregado una versión particular del sistema?
2. ¿Qué configuración de hardware y del sistema operativo se requiere para ejecutar una versión dada del sistema?
3. ¿Cuántas versiones del sistema se han creado y cuáles son sus fechas de creación?
4. ¿Qué versiones del sistema se ven afectadas si se cambia un componente particular?
5. ¿Cuántas peticiones de cambios están pendientes para una versión particular?
6. ¿Cuántos fallos declarados existen en una versión particular?

De forma ideal, la base de datos de configuraciones se integra en el sistema de gestión de las versiones utilizado para almacenar y gestionar los documentos formales del proyecto. Este enfoque, apoyado por algunas herramientas CASE integradas, hace posible vincular los cambios de forma directa con los documentos y componentes afectados por el cambio. Se da mantenimiento a los vínculos entre los documentos, como los documentos de diseño y el código del programa, con el fin de que sea relativamente fácil encontrar todo lo que debe modificarse cuando se propone un cambio.

Sin embargo, las herramientas CASE integradas para la gestión de configuraciones son caras. Muchas compañías no las utilizan, sino que mantienen su base de datos de configuraciones como un sistema independiente de su sistema de control de versiones. Los elementos de la configuración se almacenan en archivos o en el sistema de gestión de versiones como el CVS (Berliner, 1990), que se verá más adelante en este capítulo.

Esta base de datos de configuraciones almacena información de los elementos de la configuración y hace referencia a sus nombres de archivos en el sistema de gestión de versiones. Aunque éste es un enfoque relativamente económico y flexible, su desventaja es que los ele-

mentos de la configuración se pueden cambiar sin que la base de datos de configuraciones tenga conocimiento. Por lo tanto, no se puede estar seguro de que la base de datos de la configuración sea una descripción actualizada del estado del sistema.

29.2 Gestión del cambio

El cambio es un hecho en la vida de los sistemas de software grandes. Como se comentó en los primeros capítulos, las necesidades organizacionales y los requerimientos cambian durante el tiempo de vida de un sistema. Esto significa que habrá que hacer los correspondientes cambios al software del sistema. Para asegurarse de que estos cambios se hagan de forma correcta, se necesitará un conjunto de herramientas de soporte para los procedimientos de gestión de cambios.

Los procedimientos de gestión de cambios se ocupan del análisis de costes y beneficios de los cambios propuestos, aprobando aquellos cambios que merecen la pena y registrando los componentes del sistema que se tienen que cambiar. El proceso de gestión de cambios (véase la Figura 29.3) se lleva a cabo cuando el software o la documentación asociada se pone bajo el control del equipo de gestión de configuraciones.

La primera etapa del proceso de gestión del cambio es completar un formulario de solicitud de cambios (CRF) en el cual el solicitante señala los cambios requeridos en el sistema. Además de registrar los cambios requeridos, el CRF registra las recomendaciones pertinentes al cambio, los costes estimados del cambio y las fechas en las que se solicita, prueba, implementa y valida el cambio. También incluye una sección donde el analista señala cómo implementar el cambio.

En la Figura 29.4 se muestra un ejemplo de un formulario de solicitud de cambios, el cual se encuentra parcialmente rellenado. Por lo general, los formularios de solicitud de cambios se definen durante el proceso de planificación de la CM. Éste es un ejemplo de CRF que puede utilizarse en sistemas grandes y complejos. Para pequeños proyectos, recomiendo que las peticiones de cambios sean formalmente registradas, pero el CRF debe centrarse más en la descripción del cambio requerido y menos en la implementación. El ingeniero que hace el cambio decide cómo implementarlo en estas situaciones.

```
Solicitar cambios completando un formulario de solicitud de cambios
Analizar la solicitud de cambios
if cambio es válido then
    Evaluar cómo implementar el cambio
    Evaluar los costos del cambio
    Registrar la petición del cambio en una base de datos
    Remitir la petición a la oficina de control de cambios
    if cambio es aceptado then
        repeat
            Hacer cambios al software
            Registrar cambios y vincularlos a la petición de cambios asociada
            Remitir el software cambiado para aprobar la calidad
        until calidad del software sea adecuada
        Crear nueva versión del sistema
    else
        Rechazar petición de cambios
else
    Rechazar petición de cambios
```

Figura 29.3 El proceso de gestión de cambios.

Una vez emitido el formulario, se registra en la base de datos de configuraciones. Se analiza la petición de cambios para comprobar que el cambio solicitado es necesario. Algunas peticiones de cambio se deben a los malentendidos más que a los fallos del sistema y, por lo tanto, no es necesario modificar el sistema. Otras se refieren a fallos ya conocidos. Si el análisis descubre que el cambio solicitado es inválido, está duplicado o ya ha sido considerado, el cambio se rechaza. La razón del rechazo se devuelve a la persona que emitió la solicitud del cambio.

Para cambios válidos, la siguiente etapa del proceso es evaluar y asignar costes al cambio. Se comprueba el impacto del cambio en el resto del sistema. Se lleva a cabo un análisis técnico de cómo implementar el cambio. Esto implica identificar todos los componentes afectados por el cambio utilizando la base de datos de configuraciones y el código fuente. Si hacer los cambios implica hacer más cambios en otros lugares del sistema, se incrementarán los costes de implementación del cambio. A continuación, se valoraran los cambios requeridos en los módulos del sistema. Finalmente, se estima el coste del cambio, teniendo en cuenta los costes de modificar estos componentes.

El consejo de control de cambios (CCB) revisa y aprueba todas las peticiones de cambios a menos que el cambio simplemente comprenda la corrección de errores menores en los despliegues de la pantalla, las páginas web o en los documentos. El CCB considera el impacto del cambio desde el punto de vista estratégico y organizacional más que desde el punto de vista técnico. Decide si el cambio se justifica económicamente y si existen buenas razones organizacionales para aceptarlo.

La denominación «consejo de control de cambios» implica que un grupo toma las decisiones del cambio. Los proyectos militares requieren consejos de control de cambios estructurados formalmente, que incluyen a los clientes y los proveedores. Sin embargo, para proyectos pequeños o de medio tamaño, el consejo de control de cambios simplemente se compone de un gestor de proyectos más uno o dos ingenieros que no están directamente involucrados en el desarrollo del software. En algunos casos, existe sólo un revisor del cambio que aconseja si los cambios son justificables.

Proyecto: Proteus/PCL-Tools **Número:** 23/03
Solicitante del cambio: I. Sommerville **Fecha:** 1/12/02
Cambio solicitado: Cuando un componente se seleccione de una estructura, desplegar el nombre del archivo donde se almacena.

Analizador del cambio: G. Dean **Fecha de análisis:** 10/12/02
Componentes afectados: Display-Icon.Select, Display-Icon.Display
Componentes asociados: FileTable

Evaluación del cambio: Relativamente fácil de implementar puesto que se dispone de una tabla de los nombres de los archivos. Requiere el diseño e implementación de un campo de despliegue. No se requieren cambios a los componentes asociados.

Prioridad del cambio: Baja
Implementación del cambio:
Esfuerzo estimado: 0,5 días
Fecha para CCB: 15/12/02 **Fecha de decisión del CCB:** 1/2/03
Decisión del CCB: Aceptar cambio. Cambio a implementar en versión 2.1.
Implementador del cambio: **Fecha del cambio:**
Fecha de revisión para QA: **Decisión de QA:**
Fecha de revisión a CM:
Comentarios:

Figura 29.4 Un formulario de petición de cambios rellenado parcialmente.

La gestión de cambios para sistemas estándar de software debe ser manejada de una forma ligeramente diferente a los sistemas hechos a medida. En estos sistemas estándar, los clientes no están directamente implicados, por lo que un cambio en el negocio del cliente no nos afecta. Las peticiones de cambio están generalmente asociadas con errores en el sistema que han sido descubiertos durante las pruebas del sistema o por nuestros clientes una vez que ha sido entregado el mismo. Los clientes pueden enviar los informes de error a través de una página web o por e-mail. Un equipo de gestión de errores comprobará que los errores remitidos son válidos y los traducirá a peticiones de cambio formales del sistema. Los cambios tienen que ser priorizados para su implementación y los errores no pueden ser reparados si los costes de reparación son muy elevados.

Cuando se crean las nuevas versiones del sistema por medio de una construcción diaria, se utiliza un proceso de gestión del cambio sencillo. Los problemas en los cambios aún deben registrarse, pero los cambios que sólo afectan a los componentes y módulos individuales no necesitan ser evaluados de forma independiente. Se pasan directamente al desarrollador del sistema. Éste los acepta o indica por qué ya no se requieren. Los cambios que afectan a los módulos del sistema producidos por diferentes equipos de desarrollo son evaluados por alguna autoridad de control del cambio quien decide si se implementan.

En algunos métodos ágiles, como programación extrema, los clientes son directamente implicados en decidir cuándo un cambio debe ser implementado. Cuando ellos proponen un cambio en los requerimientos del sistema, trabajan con el equipo en evaluar el impacto del cambio y deciden cuándo deben tener prioridad respecto a los planes establecidos para el próximo incremento del sistema. Sin embargo, los cambios que atañen a mejoras del software se dejan a discreción de los programadores que trabajan en el sistema. La reconstrucción, donde el software es mejorado continuamente, no se ve como una sobrecarga, sino como una parte necesaria del proceso de desarrollo.

Conforme se cambian los componentes del software, se da mantenimiento al registro de los cambios realizados a cada componente. A veces éstos se denominan *historial* del componente. La mejor forma de dar mantenimiento a tales registros es por medio de una cabecera estandarizada, ubicada al inicio del componente (véase la Figura 29.5). Éste se refiere a la solicitud de cambio asociada como el cambio del software. Pueden escribirse scripts sencillos que analicen todos los componentes y procesen los historiales para generar informes de cambios para los diversos componentes. Para las páginas web se puede utilizar una aproximación similar. Para documentos publicados, los registros de cambios de cada versión son generalmente mantenidos en una portada del documento.

```
// Proyecto BANKSEC (IST 6087)
//
// BANKSEC-TOOLS/AUTH/RBAC/USER_ROLE
//
// Objeto: Regla Actual
// Autor: N. Perwaiz
// Fecha de creación: 10 de noviembre de 2002
//
// (c)Lancaster University 2002
//
// Historial de modificaciones
// Versión   Modificador   Fecha       Cambio              Razón
// 1.0       J. Jones      1/12/2002   Agregar encabezado  Remitido CM
// 1.1       N. Perwaiz    9/04/2003   Nuevo campo         Petición R07/02
```

Figura 29.5 Información de la cabecera del componente.

29.3 Gestión de versiones y entregas

La gestión de las versiones y entregas es el proceso de identificar y mantener los registros de las diversas versiones y entregas de un sistema. Los gestores de las versiones diseñan procedimientos para asegurar que las diversas versiones de un sistema se puedan recuperar cuando se requieran y que no se cambien de forma accidental por parte del equipo de desarrollo. También trabajan con los responsables de marketing y con los clientes de sistemas personalizados, en planificar las entregas y distribuciones.

Una versión de un sistema es una instancia de un sistema que difiere, de alguna manera, de otras instancias. Las nuevas versiones de un sistema tienen diferente funcionalidad, mejor rendimiento o incorporan reparaciones de los fallos del sistema. Algunas versiones son funcionalmente equivalentes pero diseñadas para diferentes configuraciones de hardware y software. Si sólo existen pequeñas diferencias entre las versiones, éstas se denominan *variantes*.

Una entrega de un sistema es una versión que se distribuye a los clientes. Cada entrega incluye nueva funcionalidad o está concebida para diferentes plataformas de hardware. Siempre existen más versiones de un sistema que las entregas, puesto que las versiones se crean dentro de una organización para el desarrollo interno o pruebas y nunca se entregan a los clientes.

En la actualidad, la gestión de versiones se apoya siempre en herramientas CASE, como se explica en la Sección 29.5. Estas herramientas administran el almacenamiento de cada versión del sistema y controlan el acceso a los componentes del sistema. Se apoyan en el sistema para llevar a cabo las ediciones. Cuando los componentes se reintroducen en el sistema, se crea una nueva versión y el sistema de gestión de versiones le asigna un identificador. A pesar de que las herramientas difieren obviamente de funcionalidades y de interfaz, la base de todas estas herramientas de soporte es la gestión de versiones.

29.3.1 Identificación de versiones

Para crear una versión particular del sistema, se tienen que especificar las versiones de cada uno de los componentes que deben incluirse en él. Dentro de un sistema de software grande, existen cientos de componentes de software, cada uno de los cuales tiene varias versiones diferentes. Debe definirse una forma no ambigua de identificar cada versión de los componentes para asegurar que se incluyen los componentes adecuados en el sistema. Sin embargo, no se puede utilizar el nombre del elemento de configuración para identificar las versiones debido a que hay diferentes versiones para cada elemento de configuración.

Existen tres técnicas básicas utilizadas para la identificación de componentes:

1. *Numeración de las versiones*. Al componente se le asigna un número de versión explícito y único. Éste es el esquema de identificación más utilizado.
2. *Identificación basada en atributos*. Cada componente tiene un nombre (como el nombre utilizado para nombrar los elementos de configuración, que no es único a lo largo de las versiones) y un conjunto asociado de atributos para cada versión del componente (Estublier y Casallas, 1994). Por lo tanto, los componentes se identifican por su nombre y por los valores de los atributos.
3. *Identificación orientada al cambio*. Cada sistema se nombra a partir de los atributos, pero también se asocia con una o más solicitudes de cambios (Munch *et al.*, 1993).

Cada versión del documento es creada en respuesta a una o más peticiones de cambios. La versión del sistema se identifica por el conjunto de los cambios implementados en el componente.

Numeración de versiones

En un esquema sencillo de numeración de versiones, al nombre del componente o del sistema se le añade un número de versión. Por lo tanto, se puede hablar de Solaris 4.3 (versión 4.3 del sistema Solaris) y la versión 1.4 del componente **getToken**. La primera versión se denomina 1.0, las subsiguientes versiones son 1.1, 1.2 y así sucesivamente. En alguna etapa, se entrega una nueva versión (versión 2.0) y el proceso comienza otra vez en la versión 2.1. El esquema es lineal y se basa en la suposición de que las versiones del sistema se crean en secuencia. Muchas herramientas de gestión de versiones (véase la Sección 29.5) como RCS (Tichy, 1985) y CVS (Berliner, 1990) permiten esta forma de identificación de versiones.

En la Figura 29.6 se muestra este enfoque y la derivación de algunas versiones diferentes del sistema a partir de versiones previas. Las flechas en este diagrama apuntan desde la versión fuente hasta una nueva versión del sistema creada a partir de la fuente. Observe que la derivación de versiones no es necesariamente lineal y las versiones con números de versión consecutivos se producen a partir de diferentes líneas base. Esto se muestra en la Figura 29.6, donde la versión 2.2 se crea a partir de la versión 1.2 en lugar de la versión 2.1. En principio, cualquier versión existente se puede utilizar como un punto de inicio para una nueva versión del sistema.

Este esquema es sencillo pero requiere una buena gestión de la información para llevar a cabo los registros de las diferentes versiones y las relaciones entre los cambios propuestos y versiones del sistema. Por ejemplo, las versiones 1.1 y 1.2 de un sistema pueden diferir en que la versión 1.2 ha utilizado una librería gráfica diferente. En consecuencia, se necesitará mantener registros en la base de datos de configuraciones que describan cada versión y por qué ha sido producida. Puede necesitarse vincular explícitamente la petición de cambios con las diferentes versiones de cada componente.

Identificación basada en atributos

Un problema fundamental con los esquemas explícitos de asignación de nombres de las versiones es que no reflejan los muchos atributos diferentes utilizados para identificar las versiones. Ejemplos de estos atributos que permiten la identificación son:

- El cliente
- El lenguaje de desarrollo

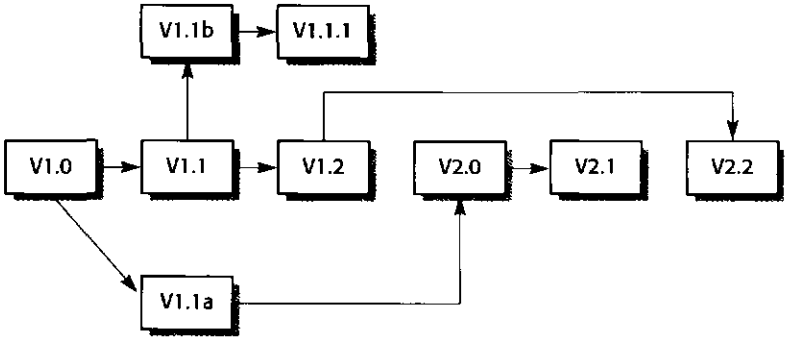


Figura 29.6 Estructura de derivación de versiones.

- El estado del desarrollo
- La plataforma de hardware
- La fecha de creación

Si cada versión es identificada por un conjunto único de atributos, es fácil agregar nuevas versiones derivadas a partir de cualquiera de las versiones existentes. Éstas se identifican utilizando un conjunto único de valores de los atributos. Comparten muchos de estos valores con sus versiones padre, por lo que se conserva la relación entre las versiones. Las versiones se recuperan especificando los valores de los atributos requeridos. Las funciones de los atributos permiten consultas como «la versión creada más recientemente», «la versión creada en fechas dadas», etcétera. Por ejemplo, la versión del sistema de software AC3D desarrollada en Java para Windows XP en enero de 2003 se identifica como:

AC3D (lenguaje = Java, plataforma = XP, fecha = Ene2003)

Usando la especificación general de los componentes en AC3D, la herramienta de gestión de versiones selecciona los componentes que tienen los atributos «Java», «XP» y «Ene2003».

La identificación basada en atributos se implementa directamente por el sistema de gestión de versiones, utilizando los atributos de los componentes mantenidos en la base de datos del sistema. Alternativamente, se puede implementar un sistema de identificación de atributos como una capa superior de un esquema de numeración de versiones oculto. La base de datos de configuraciones mantiene los vínculos entre los atributos de identificación y el sistema subyacente y las versiones del componente.

Identificación orientada al cambio

La identificación basada en atributos de las versiones del sistema elimina algunos de los problemas de la recuperación de versiones encontrados en los esquemas sencillos de numeración. Sin embargo, para recuperar una versión, se tienen que conocer los atributos asociados. Más aún, se necesita utilizar un sistema de gestión de cambios independiente para descubrir las relaciones entre las versiones y los cambios.

La identificación orientada al cambio se utiliza más para identificar los sistemas que los componentes. Las versiones de los componentes individuales están ocultas a los usuarios del sistema de la CM. Cada cambio del sistema propuesto tiene un conjunto de cambios asociado que describe los cambios realizados a los diferentes componentes del sistema para implementar este cambio. Los conjuntos de cambios se aplican en secuencia, por lo que, al menos en principio, se puede crear una versión del sistema que incorpore cualquier conjunto arbitrario de cambios. Por ejemplo, los cambios incorporados para adaptar el sistema a Linux en lugar de Solaris, seguido de los cambios necesarios para incorporar un nuevo sistema de base de datos. De igual forma, a los cambios Linux/Solaris pueden seguirlos las modificaciones para traducir la interfaz de usuario de inglés a italiano.

En la práctica, por supuesto, no es posible aplicar conjuntos arbitrarios de cambios a un sistema. Los diversos conjuntos de cambios pueden ser incompatibles, por lo que aplicar un conjunto de cambios A seguido de un conjunto de cambios D puede crear un sistema no válido. Más aún, los conjuntos de cambios pueden entrar en pugna debido a que los diversos cambios afectan al mismo código del sistema. Para afrontar estas dificultades, las herramientas de gestión de versiones que apoyan la identificación orientada a cambios permiten especificar las reglas de consistencia que delimitan las formas de combinar los conjuntos de cambios.

29.3.2 Gestión de entregas

Una entrega del sistema es una versión del sistema que se distribuye a los clientes. Los gestores de entregas del sistema son los responsables de decidir cuándo se entrega el sistema, de gestionar el proceso de creación de las entregas y de los medios de distribución y documentación de la entrega para asegurar que se puedan recuperar de la misma forma en que se distribuyeron, en caso necesario.

Una entrega del sistema no es sólo el código ejecutable del sistema. Las entregas también incluyen:

1. *Archivos de configuración*, que definen cómo configurar el sistema para instalaciones específicas.
2. *Los archivos de datos* necesarios para el funcionamiento del sistema.
3. *El programa de instalación* utilizado para ayudar a instalar el sistema en el hardware destino.
4. *La documentación electrónica y en papel* que describe al sistema.
5. *El embalaje y la publicidad asociados* diseñados para esta entrega.

Los administradores de las entregas no pueden suponer que los clientes siempre instalarán las nuevas versiones del sistema. Algunos usuarios del sistema están a gusto con una versión existente el sistema. Consideran que no vale la pena gastar en cambiar a una nueva entrega. Por lo tanto, las nuevas entregas del sistema no pueden depender de la existencia de entregas previas. Consideremos el siguiente escenario:

1. La entrega 1 de un sistema se distribuye y se pone en funcionamiento.
2. La entrega 2 requiere la instalación de nuevos archivos de datos, pero algunos clientes no necesitan los recursos de la entrega 2, por lo que conservan la entrega 1.
3. La entrega 3 requiere los archivos de datos instalados en la entrega 2 y no agrega nuevos archivos de datos.

El distribuidor de software no puede suponer que los archivos requeridos para la entrega 3 se han instalado en todos los lugares. Algunos sitios van directamente de la entrega 1 a la entrega 3, saltándose la entrega 2. Algunos sitios pueden haber modificado los archivos de datos asociados con la versión 2 para adaptarlos a circunstancias locales. Por lo tanto, los archivos de datos deben ser distribuidos e instalados con la versión 3 del sistema.

Toma de decisiones de la entrega

Preparar y distribuir una entrega del sistema es un proceso costoso, particularmente para los productos de software de mercados en masa. Si las entregas son muy frecuentes, los clientes pueden no actualizarse a las nuevas, especialmente si no son gratuitas. Si las entregas son infrecuentes, se puede perder cuota de mercado puesto que los clientes consideran sistemas alternativos. Esto, por supuesto, no es aplicable al software desarrollado específicamente para una organización. Sin embargo, para este tipo de software, las entregas infrecuentes significan un crecimiento divergente entre el software y los procesos de negocios que pretenden apoyar.

Las decisiones sobre cuándo entregar una nueva versión del sistema están dirigidas por varios factores técnicos y organizacionales, como se muestra en la Figura 29.7.

Calidad técnica	Si se reportan fallos que afecten en la forma en la que los clientes utilizan el sistema, es necesario emitir una versión para reparar el fallo. Sin embargo, los fallos menores del sistema se reparan mediante parches (a menudo distribuidos por Internet) que se aplican a las entregas actuales del sistema.
Cambios en la plataforma	Usted puede tener que crear nuevas entregas de una aplicación software cuando aparece una nueva versión del sistema operativo.
Quinta ley de Lehman (véase el Capítulo 21)	Ésta sugiere que el incremento de la funcionalidad incluida en cada versión sea aproximadamente constante. Por lo tanto, si existe una versión del sistema con funcionalidades completamente nuevas, ésta debe seguirse de una entrega de reparación.
Competencia	Se necesita una nueva versión del sistema cuando está disponible el producto competidor.
Requerimientos de marketing	El departamento de marketing de una organización puede estar interesado en tener lista la entrega en una fecha particular.
Propuestas de cambios el cliente	Para sistemas personalizados, los clientes hacen un pago por un conjunto específico de cambios en el sistema y esperan a que el sistema se entregue tan pronto como estos cambios sean implementados.

Figura 29.7 que influyen en la estrategia de entregas.

Creación de la entrega

La creación de las entregas es el proceso de crear una colección de archivos y documentación que incluyen todos los componentes de la entrega del sistema. El código ejecutable de los programas y todos los archivos de datos asociados se recogen e identifican. Se describen las configuraciones diferentes para hardware y sistemas operativos y las instrucciones para los clientes que necesiten configurar sus propios sistemas. Si se entregan manuales, las copias electrónicas deben almacenarse con el software. Se deben escribir las guías para la instalación. Finalmente, cuando toda la información está disponible, se prepara un disco de entrega para la distribución.

Actualmente, los discos ópticos (CD-ROM y DVD), que almacenan desde 600 MBytes hasta 4 GBytes, son el medio de distribución normal para la entrega del sistema. Adicionalmente, muchos productos de software también se entregan a través de Internet. Sin embargo, muchas personas, encuentran muy largo el tiempo de descarga de archivos y prefieren la distribución en CD-ROM.

La distribución de las nuevas entregas tiene asociados unos altos costes de marketing y de empaquetado, por lo que los vendedores crean usualmente nuevas entregas para nuevas plataformas o cuando se añaden funcionalidades nuevas significativamente. Ellos cobran a los usuarios por este software nuevo. Cuando se descubren problemas en una entrega, los vendedores usualmente hacen parches, que se pueden descargar vía web, para reparar el software existente.

Además de los costes de encontrar y descargar la nueva entrega, el problema es que muchos clientes nunca descubren la existencia de estos parches o no tienen los conocimientos técnicos para instalarlos. En lugar de hacer esto, pueden continuar utilizando el existente, fallando el sistema con los consiguientes riesgos para sus negocios. En algunas situaciones, en que el parche es diseñado para reparar fallos de seguridad, el riesgo de no hacer la instalación del parche puede significar que el negocio sea susceptible de ataques externos.

Documentación de las entregas

Cuando se produce la entrega de un sistema, debe estar documentada para asegurar que se puede reconstruir exactamente en el futuro. Esto es particularmente importante para los sistemas personalizados de larga vida. Los clientes utilizan una sola entrega de estos sistemas por muchos años y requieren cambios específicos para una entrega particular del software mucho después de la fecha de entrega original.

Para documentar una entrega, se tienen que registrar las versiones específicas de los componentes del código fuente utilizados para crear el código ejecutable. También se deben mantener copias del código fuente y ejecutable y todos los archivos de datos y de configuración. También se deben registrar las versiones del sistema operativo, las bibliotecas, los compiladores y otras herramientas utilizadas para construir el sistema. Éstos pueden requerirse para construir exactamente el mismo sistema en alguna fecha posterior. En estos casos, las copias de las plataformas de software y las herramientas también se almacenan en un sistema de gestión de versiones.

29.4 Construcción del sistema

La construcción del sistema es el proceso de compilar y vincular los componentes del software en un programa que se ejecuta en una configuración particular. Cuando se construye un sistema a partir de sus componentes, se tienen que hacer las siguientes preguntas:

1. ¿Todos componentes de un sistema se incluyen en las instrucciones de la construcción?
2. ¿La versión apropiada de cada componente se incluye en las instrucciones de la construcción?
3. ¿Están disponibles todos los archivos de datos requeridos?
4. Si los archivos de datos están asociados a componentes, ¿el nombre que se utiliza es el mismo que el nombre de los archivos de datos en la máquina donde se ejecutará el software?
5. ¿Están disponibles las versiones adecuadas del compilador y otras herramientas requeridas? Las versiones actuales de las herramientas de software pueden ser incompatibles con las versiones anteriores utilizadas para desarrollar el sistema.

Hoy en día, las herramientas de la CM se utilizan para automatizar el proceso de construcción del sistema. El equipo de la CM escribe una secuencia de comandos (script) que define las dependencias entre los diferentes componentes del sistema. También especifica las herramientas utilizadas para compilar y vincular los componentes del sistema. La herramienta de construcción del sistema interpreta dicha secuencia de comandos y llama a otros programas requeridos para construir el sistema ejecutable. Esto se ilustra en la Figura 29.8. En algunos entornos de programación (como los entornos de desarrollo de Java), el script para construir el sistema se crea automáticamente analizando el código fuente y estableciendo los componentes utilizados. Por supuesto, en esta situación, el nombre de los componentes almacenados tiene que ser el mismo nombre que el componente del programa.

Las dependencias entre los componentes se especifican en la secuencia de comandos de construcción, por lo que el sistema de construcción decide cuándo los componentes deben recompilarse y cuándo se puede reutilizar el código objeto existente. En muchas herramientas,

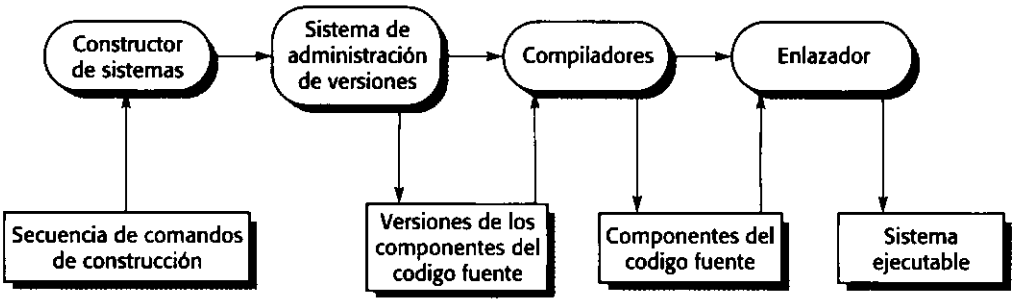


Figura 29.8 Construcción del sistema.

las dependencias de la secuencia de comandos de construcción se especifican como dependencias entre los archivos en los cuales los componentes de código fuente se almacenan. Sin embargo, cuando existen múltiples archivos de código fuente representando a las diferentes versiones de los componentes, algunas veces no se ve claramente qué archivos fuente se utilizaron para derivar los componentes del código objeto. Esta confusión aparece cuando la correspondencia entre los archivos de código fuente y objeto comparten el mismo nombre pero diferente sufijo (por ejemplo, .c y .o). La única forma de evitar este problema es que las herramientas de gestión de versiones y de construcción del sistema estén integradas.

29.5 Herramientas CASE para gestión de configuraciones

Los procesos de gestión de configuraciones por lo general están estandarizados e involucran la aplicación de procedimientos predefinidos. Requieren la gestión cuidadosa de grandes cantidades de datos, y la atención a los detalles es esencial. Cuando se construye un sistema a partir de las versiones de los componentes, un simple error en la gestión de configuraciones puede implicar que el software no trabaje de forma adecuada. En consecuencia, las herramientas CASE de apoyo son esenciales para la gestión de configuraciones, y desde los 70 se ha producido un gran número de diferentes herramientas que abordan diferentes áreas de la gestión de configuraciones.

Estas herramientas pueden ser combinadas para crear entornos de trabajo de gestión de configuraciones que soporten todas las actividades CM. Hay dos tipos de entornos de trabajo CM:

1. *Entornos de trabajo abiertos.* Las herramientas para cada etapa del proceso CM son integradas de acuerdo con procedimientos organizacionales estándar. Hay muchas herramientas CM comerciales y open-source disponibles para propósitos específicos. La gestión de cambios puede llevarse a cabo con herramientas de seguimiento (bug-tracking) como Bugzilla, la gestión de versiones a través de herramientas como RCS (Tichy, 1985) o CVS (Berliner, 1990), y la construcción del sistema con herramientas como Make (Feldman, 1979; Oram y Talbott, 1991) o Imake (Dubois, 1996). Todas estas herramientas son open-source y están disponibles de forma gratuita.
2. *Entornos integrados.* Estos entornos ofrecen facilidades integradas para gestión de versiones, construcción del sistema o seguimiento de los cambios. Por ejemplo, el proceso de gestión de Cambios Unificado de Rational se basa en un entorno CM integrado que incorpora ClearCase (White, 2000) para la construcción y gestión de versiones

del sistema y ClearQuest para el seguimiento de los cambios. Las ventajas de los entornos CM integrados es que el intercambio de datos es sencillo, y que el entorno incluye una base de datos CM integrada. Los entornos integrados SCM provienen de sistemas antiguos como Lifespan (Whitgift, 1991) para gestión de cambios y DSEE (Leblang y Chase, 1987) para gestión de versiones y construcción del sistema. Sin embargo, los entornos integrados CM son complejos y costosos, y muchas organizaciones prefieren utilizar herramientas individuales más simples y económicas.

Muchos sistemas grandes son desarrollados desde diferentes lugares, y necesitan herramientas SCM que soporten el trabajo desde múltiples lugares con múltiples almacenes de datos para los elementos de configuración. Mientras muchas herramientas SCM están diseñadas para trabajar desde un único lugar, otras como CVS facilitan el trabajo desde múltiples sitios.

29.5.1 Apoyo a la gestión de cambios

Cada persona involucrada en el proceso de gestión de cambios es responsable de alguna actividad. Una vez que completa esta actividad pasa los formularios y elementos de configuración asociados a otra persona. La naturaleza de procedimiento de este proceso significa que un cambio en el modelo del proceso se diseña e integra con un sistema de gestión de versiones. Entonces este modelo se interpreta para que los documentos correctos se pasen a la persona indicada en el momento justo.

Hay varias herramientas de gestión de cambios disponibles, desde herramientas relativamente simples, open-source como Bugzilla hasta sistemas absolutamente integrados como Racional ClearQuest. Estas herramientas proveen alguna o todas de las siguientes características de apoyo al proceso:

1. Un editor de formularios que permite cambiar los formularios propuestos a crear y rellenar por las personas que hacen las peticiones de cambios.
2. Un sistema de flujo de trabajo que permiten al equipo de la CM especificar las personas que deben procesar las solicitudes de peticiones de cambio y el orden del procesamiento. Este sistema también pasa de forma automática los formularios a las personas indicadas en el momento justo e informa a los miembros relevantes del equipo del progreso del cambio. El correo electrónico se utiliza para suministrar actualizaciones del progreso a aquellos involucrados en el proceso.
3. Una base de datos de cambios que se utiliza para gestionar todas las propuestas de cambios y que puede vincularse al sistema de gestión de versiones. Por lo general, se proveen las características de consulta que permiten al equipo de la CM encontrar propuestas específicas de cambios.
4. Un sistema de gestión de informes de cambios que genera informes sobre el estado de la peticiones de cambio recibidas.

29.5.2 Soporte para gestión de versiones

La gestión de versiones implica gestionar grandes cantidades de información para asegurar que los cambios en el sistema se registren y controlen. Las herramientas de gestión de versiones controlan un repositorio de elementos de configuración donde el contenido de ese repositorio es inmutable (no cambia). Para trabajar sobre un elemento de la configuración, debe extraerse del re-

positorio y colocarlo en un directorio de trabajo. Después de hacer los cambios en el software, se introducirá de nuevo en el repositorio, creándose automáticamente una nueva versión. Todos los sistemas de gestión de versiones proveen un conjunto básico de capacidades comparables aunque algunos son más sofisticados que otros. Ejemplos de estas capacidades son:

1. *Identificación de versiones y entregas.* A las versiones gestionadas se les asignan identificadores cuando se introducen en el sistema. Varios sistemas permiten los diferentes tipos de identificación de versiones tratados en la Sección 29.3.1.
2. *Gestión del almacenamiento.* Para reducir el espacio de almacenamiento requerido por las diferentes versiones, los sistemas de gestión de versiones proveen las características de gestión del almacenamiento donde las versiones se describen por sus diferencias a partir de alguna versión maestra. Las diferencias entre las versiones se representan con una *delta* que encapsula las instrucciones requeridas para reconstruir la versión asociada del sistema. Esto se ilustra en la Figura 29.9, que muestra cómo se aplican las deltas inversamente a la última versión de un sistema para reconstruir las versiones anteriores.
3. *Registro del historial del cambio.* Todos los cambios realizados al código de un sistema o componente se registran y son listados. En algunos sistemas, estos cambios se utilizan para seleccionar una versión particular del sistema.
4. *Desarrollo independiente.* Las diferentes versiones de un sistema se pueden desarrollar en paralelo y cada versión cambia de forma independiente. Por ejemplo, la entrega 1 se modifica después del desarrollo de la entrega 2 agregando nuevas deltas de nivel 1. El sistema de gestión de versiones mantiene un registro de los componentes que han sido extraídos para su edición y asegura que no interfieran los cambios que diferentes desarrolladores hayan hecho a los mismos componentes. Algunos sistemas sólo permiten que se extraiga una instancia de un componente para su edición; otros resuelven los conflictos potenciales cuando los componentes editados se reintroducen en el sistema.
5. *Apoyo al proyecto.* El sistema puede soportar múltiples proyectos así como múltiples archivos. En sistemas de apoyo a proyectos, como CVS, es posible introducir y extraer todos los ficheros asociados a un proyecto en lugar de tener que trabajar con un solo fichero a la vez.

29.5.3 Apoyo a la construcción del sistema

La construcción de sistemas es un proceso intensivo de computación. Compilar y vincular todos los componentes de un sistema grande puede llevar varias horas. Puede haber cientos de archivos involucrados con la consecuente posibilidad de errores humanos si éstos se compilan y vinculan de forma manual. Las herramientas de construcción de sistemas automatizan

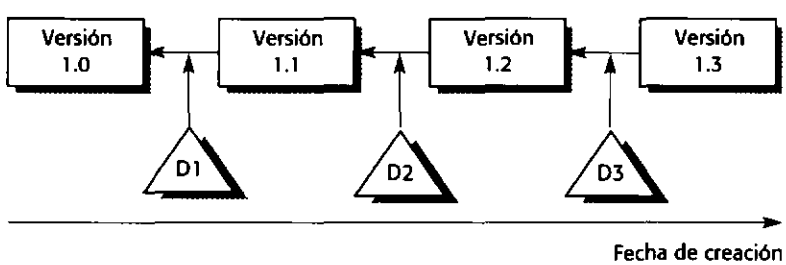


Figura 29.9 Versiones basadas en delta.

el proceso de construcción para reducir los errores humanos potenciales y, donde sea posible, disminuir el tiempo requerido para la construcción del sistema.

Las herramientas de construcción del sistema pueden ser independientes como las derivadas de la utilidad MAKE de Unix (Oran y Talbott, 1991) o integradas con las herramientas de gestión de versiones. Las características suministradas por las herramientas CASE de construcción de sistemas son:

1. *Una dependencia del lenguaje de especificación o del intérprete asociado.* Las dependencias de los componentes se describen y se disminuye la recompilación. Esto se explica más adelante en esta sección.
2. *Selección de herramientas y apoyo a la instanciación.* Se especifican e instancian, como se requiera, los compiladores y otras herramientas de procesamiento utilizadas para procesar los archivos de código fuente.
3. *Compilación distribuida.* Algunos constructores de sistemas, especialmente aquellos que son parte de los sistemas integrados de la CM, permiten la compilación distribuida en redes. En lugar de que todas las compilaciones se lleven a cabo en una sola máquina, los constructores de sistemas buscan los procesadores desocupados sobre la red y seleccionan varios compiladores en paralelo. Esto reduce notablemente el tiempo requerido para construir un sistema.
4. *Gestión de los objetos derivados.* Los objetos derivados son objetos que se crean a partir de otros objetos fuente. La gestión de los objetos derivados vincula el código fuente y los objetos derivados y sólo vuelve a derivar un objeto cuando se requiere por los cambios del código fuente.

Gestionar los objetos derivados y disminuir la recompilación se explica mejor utilizando un ejemplo sencillo. Consideremos una situación donde un programa denominado **comp** se crea a partir de los módulos objeto **scan.o, syn.o, sem.o** y **cgen.o**. Para cada módulo objeto, existe un módulo del código fuente (**scan.c, syn.c, sem.c** y **cgen.c**). Un archivo de declaraciones denominado **defs.h** es compartido por **scan.c, syn.c** y **sem.c** (véase la Figura 29.10). En la Figura 29.10 las flechas significan «depende de». Por lo tanto, **comp** depende de **scan.o, syn.o, sem.o** y **cgen.o**, **scan.o** dependen de **scan.c**, etcétera.

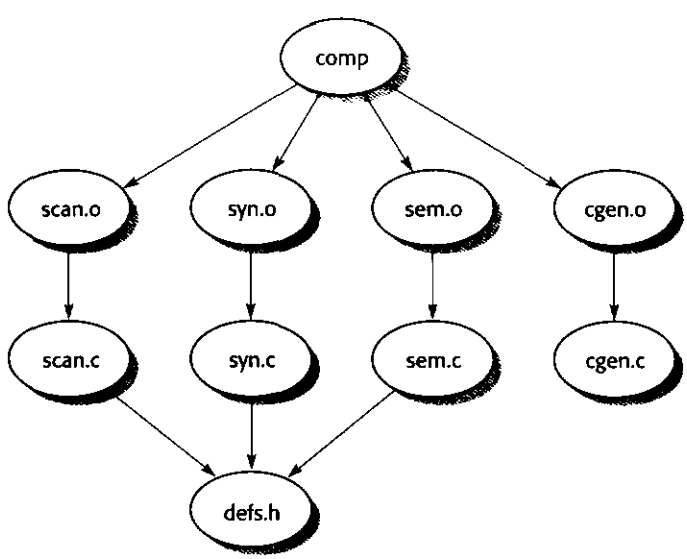


Figura 29.10
Dependencias entre componentes.

Si **scan.c** cambia, la herramienta de construcción del sistema puede detectar que el objeto derivado **scan.o** debe reconstruirse y llamar al compilador de C para compilar **scan.c** con el fin de crear un nuevo objeto derivado, **scan.o**.

Después la herramienta de construcción utiliza el vínculo de dependencia entre **comp** y **scan.o** para detectar que **comp** debe reconstruirse vinculando **scan.o, syn.o, sem.o** y **cgen.o**. El sistema puede detectar que los otros componentes del código objeto no cambiaron, por lo que no es necesaria la recompilación del código fuente asociado.

Algunos constructores de sistemas utilizan el archivo de la fecha de modificación como atributo clave para decidir si se requiere la recompilación. Si un archivo de código fuente se modifica después de su correspondiente archivo de código objeto, entonces este último se reconstruye. Esto normalmente significa que existe un objeto derivado sólo para la versión del código fuente más reciente. Cuando se crea una nueva versión del código objeto la versión anterior de éste se pierde.

Otros sistemas utilizan un enfoque más sofisticado para derivar la gestión de los objetos. Agregan objetos derivados al identificador de la versión del código fuente utilizado para generar estos objetos. Dentro de los límites de la capacidad de almacenamiento, conservan todos los objetos derivados. Por lo tanto, generalmente es posible recuperar el código objeto de todas las versiones de los componentes del código fuente sin recompilación.

PUNTOS CLAVE

- **La gestión de configuraciones es la gestión de los cambios en el sistema. Cuando se mantiene un sistema, el papel del equipo de CM es asegurar que los cambios se incorporen de forma controlada.**
- **En un proyecto grande, se establece y utiliza un esquema formal de nombres de documentos como una base para mantener el registro de las diferentes versiones de todos los documentos del proyecto.**
- **El equipo de CM se apoya en una base de datos de configuraciones que registra la información de los cambios en el sistema y de las peticiones de cambio pendientes. Los proyectos deben tener algún medio formal de petición de cambios al sistema.**
- **Cuando se fija un esquema de gestión de configuraciones, se establece un esquema consistente de identificación de versiones. Éstas se identifican por el número de versión, por un conjunto de atributos o por los cambios propuestos e implementados en el sistema.**
- **Las entregas de los sistemas incluyen el código ejecutable, los archivos de datos, los archivos de configuración y la documentación. La gestión de las entregas comprende tomar decisiones sobre cuándo entregar un sistema, preparar toda la información para la distribución y la documentación de cada entrega del sistema.**
- **La construcción de sistemas es el proceso de ensamblar los componentes del sistema en un programa ejecutable para que opere en un sistema informático específico.**
- **Existen herramientas CASE para apoyar las actividades de gestión de la configuración. Éstas comprenden herramientas como CVS para gestionar las versiones del sistema, herramientas para la gestión del cambio y herramientas para la construcción de sistemas.**
- **Las herramientas CASE para la CM pueden ser herramientas autónomas que permiten gestionar el cambio, gestionar las versiones y la construcción de sistemas o pueden ser sistemas integrados que proveen una sola interfaz para las tareas de CM.**

LECTURAS ADICIONALES

Configuration Management Principles and Practice. Es un estudio extenso que incluye los estándares y las aproximaciones tradicionales de la CM, así como aproximaciones de la CM que son más adecuadas para los procesos modernos como el desarrollo ágil de software. (A. M. J. Hass, 2002, Addison-Wesley.)

«A layered architecture for uniform version management». Este artículo analiza las diferentes aproximaciones a la gestión de versiones del software y propone un modelo básico que puede adaptarse a todas ellas. Incluye un buen informe sobre trabajos en esta área. [B. Westfechel *et al., IEEE Transactions on Software Engineering,* 27 (12), diciembre de 2001.]

«Software Configuration Management: A roadmap». Este artículo presenta una perspectiva sobre la evolución SMC e identifica las investigaciones nuevas en esta área. (J. Estublier, *Proc. Int. Conf. on Software Engineering,* 2000. IEEE Press.)

Trends in Software: Configuration Management. Es una colección de artículos sobre diferentes aspectos de la gestión de la configuración escritos por autores que son investigadores y profesionales en este campo. Es una buena introducción para estudiantes y profesionales interesados en leer temas de CM avanzados. [W. Tichy (ed.), 1995, John Wiley and Sons.]

EJERCICIOS

29.1 Explique por qué no se debe utilizar el título de un documento para identificar dicho documento en un sistema de gestión de la configuración. Sugiera un estándar para un esquema de identificación de documentos que se pueda utilizar para todos los proyectos de una organización.

29.2 Utilizando el enfoque orientado a objetos (véase el Capítulo 8), diseñe un modelo de una base de datos de configuración que registre la información de los componentes del sistema, las versiones, las entregas y los cambios. Algunos requerimientos del modelo de datos son los siguientes:

- Debe ser posible recuperar todas las versiones o una versión específica de un componente.
- Debe ser posible recuperar la última versión de un componente.
- Debe ser posible saber qué peticiones de cambios se implementaron en una versión específica del sistema.
- Debe ser posible saber qué versiones de los componentes se incluyeron en una versión específica de un sistema.
- Debe ser posible recuperar una versión particular de un sistema según su fecha de entrega o de acuerdo con los clientes a quienes se les hizo la entrega.

29.3 Utilizando un diagrama de flujo de datos, describa un procedimiento de gestión de cambios que se pueda utilizar en una organización grande que se dedique a desarrollar software para clientes externos. Los cambios se pueden sugerir de fuentes externas o internas.

29.4 ¿Cómo el uso de sistemas de gestión de versiones basado en proyectos como CVS simplifica el proceso de gestión de versiones?

29.5 Explique por qué en un sistema de identificación de versiones basada en atributos es más fácil descubrir todos los componentes utilizados en una versión específica del sistema.

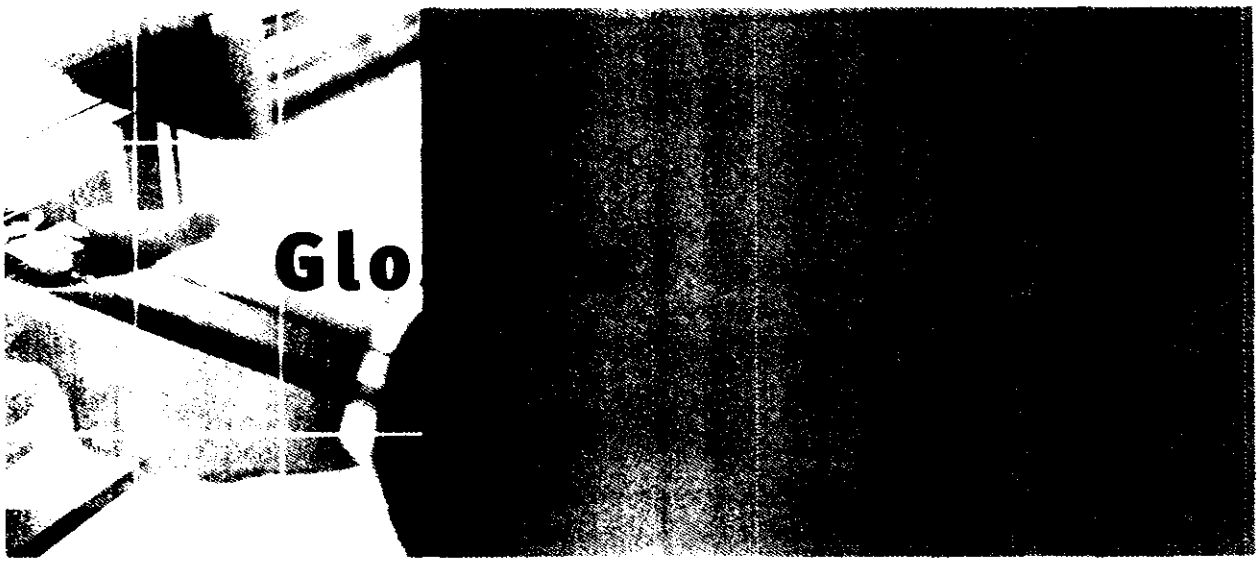
29.6 Describa las dificultades que pueden aparecer en la construcción de sistemas. Indique los problemas particulares que surgen cuando se construye un sistema sobre una computadora y el programa se va a ejecutar en otra máquina específica.

29.7 Con referencia a la construcción de sistemas, explique por qué algunas veces es necesario conservar los ordenadores obsoletos sobre las cuales se desarrollaron los sistemas de software.

29.8 Un problema común en la construcción de sistemas ocurre cuando los nombres de los archivos físicos se incorporan al código del sistema y la estructura del archivo implicada en esos nombres difiere de la estructura de la máquina en que se ejecutará. Redacte un conjunto de guías para el programador que ayuden a evitar este y otros problemas en la construcción de sistemas.

29.9 Describa cinco factores que deben tener en cuenta los ingenieros durante el proceso de construcción y entrega de un sistema de software grande.

29.10 Describa dos formas en las cuales las herramientas de construcción de sistemas pueden optimizar el proceso de construcción de una versión de un sistema a partir de sus componentes.



Ada
Lenguaje de programación que fue desarrollado por el Departamento de Defensa de los Estados Unidos como un lenguaje estándar para el desarrollo de software militar. Está basado en las investigaciones en los lenguajes de programación a partir de los años 70 e incluye construcciones como los tipos abstractos de datos y soporte para concurrencia. Todavía se utiliza en grandes sistemas aeroespaciales militares complejos.

análisis estático
Análisis basado en herramientas del código fuente de un programa para descubrir errores y anomalías. Las anomalías como las asignaciones sucesivas a una variables sin un uso intermedio pueden ser errores de programación.

arquitectura cliente-servidor
Modelo arquitectónico para sistemas distribuidos en el que la funcionalidad del sistema se ofrece como un conjunto de servicios proporcionados por un servidor. Éstos son accedidos por computadoras cliente que hacen uso de los servicios. Variantes de este enfoque, como las arquitecturas cliente-servidor de tres capas, utilizan múltiples servidores.

arquitectura de referencia
Arquitectura genérica del sistema que es una arquitectura ideal que incluye todas las características que los sistemas podrían incorporar. Constituye un modo de informar a los diseñadores sobre la estructura general de esa clase de sistemas.

arquitectura software
Modelo de la estructura y organización fundamental de un sistema software.

banco de trabajo CASE
Conjunto integrado de herramientas CASE que trabajan conjuntamente para apoyar una actividad del proceso importante como el diseño del software o la gestión de la configuración.

C
Lenguaje de programación que fue originalmente desarrollado para ayudar a implementar el sistema Unix. C es un lenguaje de implementación de sistemas de relativamente bajo nivel que permite el acceso al hardware del sistema y que puede ser compilado a un código eficiente. Todavía se utiliza ampliamente para la programación sistemas de bajo nivel.

C++
Lenguaje de programación orientado a objetos. C es un subconjunto de C++.

caso de confiabilidad
Documento estructurado que se utiliza para apoyar las afirmaciones efectuadas por un desarrollador del sistema sobre la confiabilidad de un sistema.

caso de seguridad
Argumento estructurado de que un sistema es seguro. Normalmente es requerido por reguladores tales como los reguladores de la seguridad nuclear.

caso de uso
Especificación de un tipo de interacción con un sistema.

ciclo de vida del software
Utilizado a menudo como otro nombre para el proceso del software. Originalmente acuñado para referirse al modelo en cascada del proceso del software.

clase de objetos
Una clase de objetos define los atributos y operaciones de los objetos. Los objetos se crean en tiempo de ejecución mediante la instanciación de la definición de la clase. El nombre de la clase de objetos se puede utilizar como un nombre de tipo en algunos lenguajes orientados a objetos.

CMMI
Enfoque integrado para el modelado de madurez de la capacidad del proceso. Apoya los modelados de madurez discretos y continuos e integra sistemas y modelos de madurez de los procesos de la ingeniería del software.

código de ética y práctica profesional
Conjunto de pautas que exponen el comportamiento ético y profesional esperado de los ingenieros del software. Fue definido por las sociedades profesionales principales de los Estados Unidos (la ACM y la IEEE) y define el comportamiento ético conforme a ocho títulos: público, cliente y empleador, producto, juicio, gestión, colegas, profesión y personal.

COM+
Un modelo de componentes diseñado para su uso en plataformas Microsoft.

Common Request Broker Architecture (CORBA)
Conjunto de estándares propuesto por el OMG que define un modelo de objetos distribuido y las comunicaciones de los objetos.

componente
Unidad de software independiente y desplegable que se ha definido completamente y a la que se accede a través de un conjunto de interfaces.

confiabilidad
La confiabilidad de un sistema es una propiedad total que tienen en cuenta la seguridad del sistema, la fiabilidad, la disponibilidad, la protección y otros atributos. La confiabilidad de un sistema refleja el grado en el cual los usuarios pueden confiar en el sistema.

construcción del sistema
Proceso de compilar los componentes o unidades que forman un sistema y enlazarlos con otros componentes para crear un programa ejecutable. La construcción del sistema está normalmente automatizada de modo que se minimiza la recompilación. Esta automatización puede ser incorporada a un sistema de procesamiento de lenguajes (como en Java) o puede implicar herramientas CASE para apoyar la construcción del sistema.

Constructive Cost Modelling (COCOMO)
Quizás el modelo algorítmico de estimación de costes más conocido.

control de la calidad (QC)
Proceso de asegurar que un equipo de desarrollo de software sigue los estándares de calidad.

desarrollo del software orientado a aspectos
Enfoque para el desarrollo de software que combina el desarrollo generativo y el basado en componentes. Se identifican los intereses compartidos en un programa y la implementación de estos intereses se define como aspectos. Un programa se encarga entonces de entrelazar los aspectos en los lugares apropiados en el programa.

desarrollo incremental
Enfoque para el desarrollo de software en el que éste se entrega y utiliza en incrementos.

desarrollo iterativo
Enfoque para el desarrollo de software en el que se entrelazan los procesos de especificación, diseño, programación y pruebas.

desarrollo orientado a objetos (OO)
Enfoque para el desarrollo de software en el que las abstracciones fundamentales en el sistema son objetos independientes. Se utiliza el mismo tipo de abstracción durante la especificación, diseño y desarrollo.

desarrollo rápido de aplicaciones (RAD)
Enfoque para el desarrollo de software dirigido a la entrega rápida de éste. A menudo implica el uso de la programación de bases de datos y herramientas de apoyo al desarrollo como los generadores de informes y pantallas.

detección de defectos
Utilización de procesos y comprobaciones en tiempo de ejecución para detectar y eliminar los defectos en un programa antes de que éstos causen un fallo de funcionamiento del sistema.

diagrama de secuencia
Diagrama que muestra la secuencia de interacciones necesarias para completar alguna operación. En UML, los diagramas de secuencias se pueden asociar con los casos de uso.

dinámica de evolución de los programas
Estudio de las formas en las que cambia un sistema software que se desarrolla.

diseño de interfaces de usuario
Proceso de diseñar el modo en el que los usuarios del sistema acceden a la funcionalidad del sistema y la forma en la que se visualiza la información producida por el sistema.

disponibilidad
Preparación de un sistema para entregar servicios cuando se le soliciten. La disponibilidad normalmente se expresa como un número decimal; así una disponibilidad de 0,999 significa que el sistema puede entregar servicios durante 999 de cada 1.000 unidades de tiempo.

dominio
Problema o área de negocio específico donde son utilizados los sistemas software. Ejemplos de dominios son el control en tiempo real, el procesamiento de datos de negocio y la conmutación de telecomunicaciones.

elemento de configuración
Unidad legible por la máquina, como un documento o un fichero de código fuente, que es susceptible de cambiar y donde los cambios tienen que ser controlados por un sistema de gestión de la configuración.

enterprise java beans (EJB)
Modelo de componentes basado en Java.

entrega
Versión de un sistema software que se pone a disposición de los clientes del sistema.

escenario
Descripción de una forma típica en la que se utiliza un sistema o un usuario lleva a cabo alguna actividad.

especificación formal, algebraica
Método de especificación matemática de sistemas en el que un sistema o componente se especifica definiendo las relaciones entre las operaciones definidas en sus interfaces externas.

especificación formal, basada en modelos
Método de especificación matemática de sistemas en el que un sistema o componente se especifica definiendo las precondiciones, postcondiciones e invariantes que se aplican al estado del sistema.

etnografía
Técnica basada en la observación que puede ser utilizada en la obtención y análisis de requerimientos. El etnógrafo se sumerge en el entorno del usuario y observa sus hábitos de trabajo cotidianos. A partir de estas observaciones se pueden deducir requerimientos para apoyo al software.

evitación de defectos
Desarrollo del software de tal modo que no se introduzcan defectos en ese software.

familia de aplicaciones
Conjunto de programas de aplicaciones software que tienen una arquitectura común y una funcionalidad genérica. Éstos se pueden adaptar a las necesidades específicas de los clientes modificando componentes y parámetros de los programas.

fiabilidad
Capacidad de un sistema para entregar los servicios como se especifican. La fiabilidad se puede especificar cuantitativamente como la probabilidad de que ocurra un fallo de funcionamiento o como la tasa de ocurrencia de éstos.

garantía de la calidad (QA)
Proceso general de definir cómo lograr la calidad del software y cómo la organización de desarrollo conoce el nivel de calidad requerido en el software.

generador de programas
Programa que genera otro programa a partir de una especificación abstracta de alto nivel. El generador incorpora conocimientos que se reutilizan en cada actividad de generación.

gestión de la configuración
Proceso de gestionar los cambios a un producto software que se desarrolla. La gestión de la configuración implica la planificación de la configuración, la gestión de versiones, la construcción del sistema y la gestión del cambio.

gestión de requerimientos
Proceso de gestionar los cambios en los requerimientos para asegurar que los cambios efectuados son correctamente analizados e implementados en el sistema.

gestión de riesgos
Proceso de identificar los riesgos, evaluar su gravedad, planificar las medidas a adoptar si se presenta el riesgo y supervisar el software y los procesos software para los riesgos.

gráfico de actividades (PERT)
Gráfico utilizado por los gestores de proyectos para mostrar las dependencias entre las tareas que se tienen que completar. El gráfico muestra las tareas, el tiempo esperado para completarlas y las dependencias entre ellas. El camino crítico es el camino más largo (en función del tiempo requerido para completar las tareas) a lo largo del gráfico de actividades. El camino crítico define el tiempo mínimo requerido para completar el proyecto.

gráfico de barras (Gantt)
Gráfico utilizado por los gestores de proyectos para mostrar las tareas del proyecto, la agenda asociada con estas tareas y las personas que trabajarán en ellas. Muestra las fechas de comienzo y finalización de las tareas y la asignación de personal contra una línea de tiempo.

herramienta CASE
Herramienta software, como un editor del diseño o un depurador de programas, utilizada para apoyar una actividad en el proceso de desarrollo del software.

ingeniería de sistemas
Proceso que trata de la especificación de un sistema, la integración de sus componentes y las pruebas de que el sistema cumple sus requerimientos. La ingeniería de sistemas no sólo trata el sistema software, sino el sistema socio-técnico entero: software, hardware y procesos operativos.

Ingeniería del Software Asistida por Computadora (CASE)
Proceso de desarrollar software utilizando ayuda automatizada.

ingeniería del software basada en componentes (CBSE)
Desarrollo de software a partir de la composición de componentes independientes y desplegables.

ingeniería del software de sala limpia
Enfoque para el desarrollo de software en el que el objetivo es evitar introducir defectos en el software (por analogía con una sala limpia utilizada en la fabricación de semiconductores). El proceso implica la especificación formal del software, la transformación estructurada de una especificación a un programa, el desarrollo de argumentos de la corrección y pruebas estadísticas de programas.

inspección de programas
Proceso de verificación en el que un grupo de revisores examina un programa, línea por línea, con el objetivo de detectar errores.

Interfaz de Programación de Aplicaciones (API)
Interfaz, generalmente especificada como un conjunto de operaciones, definida por un programa de aplicación que permite acceder a la funcionalidad del programa. Esto significa que no sólo se puede acceder a esta funcionalidad a través de la interfaz de usuario, sino que otros programas pueden utilizarla directamente.

interfaz
Especificación de los atributos y operaciones asociados con un componente software. La interfaz es utilizada como el medio de tener acceso a la funcionalidad del componente.

ISO 9000
Estándar para los procesos de gestión de calidad definido por la Organización Internacional de Normalización (ISO).

Java
Lenguaje de programación orientado a objetos que fue diseñado por Sun con el objetivo de la independencia de la plataforma.

Lenguaje de Modelado Unificado (UML)
Lenguaje gráfico utilizado en el desarrollo orientado a objetos que incluye varios tipos de modelos del sistema que proporcionan distintas vistas de un sistema. UML se ha convertido en un estándar *de facto* para el modelado orientado a objetos.

lenguaje de restricciones de objetos (OCL)
Lenguaje que forma parte de UML, utilizado para definir predicados que se aplican a las clases de objetos e interacciones en un modelo UML.

Lenguaje Estructurado de Consultas (SQL)
Lenguaje estándar utilizado para la programación de bases de datos relacionales.

línea de productos software
Véase familia de aplicaciones.

mantenimiento
Proceso de hacer cambios en un sistema después de que esté en funcionamiento.

marcos de trabajo de aplicaciones
Estructura genérica en algún dominio específico que puede formar la base de una familia de aplicaciones. Los marcos de trabajo de aplicaciones generalmente se implementan como un conjunto clases concretas y abstractas especializadas e instanciadas para crear una aplicación.

mejora de procesos
Proceso de hacer cambios a un proceso con el objetivo de hacerlo más previsible o mejorar la calidad de sus salidas. Por ejemplo, si su objetivo es reducir el número de defectos en el software entregado, podría mejorar el proceso añadiendo nuevas actividades de validación.

método estructurado
Método de diseño de software que define los modelos del sistema que se deben desarrollar, las reglas y pautas que se deben aplicar a estos modelos y un proceso a seguir en el desarrollo del diseño.

métodos ágiles
Métodos de desarrollo de software dirigidos a la entrega rápida del mismo. El software se desarrolla y entrega en incrementos, y se minimiza el proceso de documentación y la burocracia.

métodos formales
Métodos de desarrollo de software basados en enfoques matemáticamente rigurosos que modelan el software utilizando construcciones matemáticas formales como predicados y conjuntos.

métrica software
Atributo de un sistema o proceso software que se puede medir o expresar numéricamente. Las métricas de procesos son atributos del proceso como el tiempo necesario para completar una tarea; las métricas de productos son atributos del software mismo como el tamaño o la complejidad.

middleware
Infraestructura software en un sistema distribuido. Ayuda a gestionar las interacciones entre las entidades distribuidas en el sistema y las bases de datos del mismo. Ejemplos de middleware son un intermediario de peticiones de objetos y un sistema de gestión de transacciones.

modelado algorítmico de costes
Enfoque para la estimación del coste del software en el que se utiliza una fórmula para estimar el coste del proyecto. Los parámetros en la fórmula son atributos del proyecto y el software mismo.

modelado del aumento de la fiabilidad
Desarrollo de un modelo de cómo cambia la fiabilidad de un sistema (se espera que mejore) conforme éste se prueba y eliminan los defectos de los programas.

modelo de componentes CORBA
Modelo de componentes diseñado para su uso para plataformas CORBA.

modelo de componentes
Conjunto de estándares para la implementación, documentación y utilización de componentes. Éstos cubren las interfaces específicas que pueden ser proporcionadas por un componente, el nombrado de componentes, la interoperatividad de los componentes y la composición de éstos. Los modelos de componentes proporcionan la base al middleware para soportar la ejecución de componentes.

Modelo de Madurez de la Capacidad del Personal (P-CMM)
Modelo de madurez del proceso que refleja cómo de efectiva es una organización gestionando las habilidades, formación y experiencia del personal de esa organización.

modelo de madurez del proceso
Modelo del grado en el que un proceso incluye buenas prácticas y capacidades de medida y reflexivas que están orientadas a la mejora de procesos.

modelo de objetos
Modelo de un sistema software que se estructura y organiza como un conjunto de clases de objetos y las relaciones entre estas clases. Pueden existir varias perspectivas diferentes en el modelo, como una perspectiva del estado y una perspectiva de la secuencia.

modelo de procesos
Representación abstracta de un proceso. Los modelos de procesos pueden ser representados desde varias perspectivas y mostrar las actividades implicadas en un proceso, los objetos utilizados en el proceso, las restricciones que se aplican al proceso y los roles de las personas involucradas en el proceso.

modelo del dominio
Definición de abstracciones del dominio como políticas, procedimientos, objetos, relaciones y eventos. Sirve de base de conocimiento sobre alguna área del problema.

modelo en cascada
Modelo del proceso del software en el que existen diferentes etapas de desarrollo: especificación, diseño, implementación, pruebas y mantenimiento. En principio, se debe completar una etapa antes de que se pueda avanzar a la siguiente. En la práctica, existe iteración entre las etapas.

modelo en espiral
Modelo de un proceso de desarrollo donde el proceso se representa como una espiral en la que cada vuelta de la espiral incorpora las diferentes etapas en el proceso. Si se pasa de una vuelta de la espiral a otra, se repiten todas las etapas del proceso.

Object Management Group (OMG)
Grupo de compañías formado para desarrollar estándares para el desarrollo orientado a objetos. Ejemplos de estándares promovidos por el OMG son CORBA, UML y MDA.

ocultación de información
Utilización de construcciones de lenguajes de programación para ocultar la representación de las estructuras de datos y controlar el acceso externo a estas estructuras.

patrón de diseño
Solución probada a un problema común que capta las experiencias y buenas prácticas de una forma que puede ser reutilizada. Es una representación abstracta que se puede instanciar de varias formas.

plan de calidad
Plan que define los procesos y procedimientos de calidad que se deben utilizar. Implica seleccionar e instanciar estándares para productos y procesos y definir los atributos de la calidad requeridos del sistema.

principios de diseño de las interfaces de usuario
Conjunto de principios que expresan buenas prácticas para el diseño de interfaces de usuario.

proceso del software
Conjunto relacionado de actividades y procesos implicados en el desarrollo y evolución de un sistema software.

Proceso Unificado de Rational (RUP)
Modelo de proceso del software genérico que presenta el desarrollo del software como una actividad iterativa de cuatro fases que son inicio, elaboración, construcción y transición. La fase de inicio establece un caso de negocio para el sistema, la fase de elaboración define la arquitectura, la de construcción implementa el sistema y la de transición utiliza el sistema en el entorno del cliente.

programación extrema (XP)
Método ágil de desarrollo de software que incluye prácticas como los requerimientos basados en escenarios, el desarrollo previamente probado y la programación en parejas.

propiedad emergente
Propiedad que sólo se hace evidente una vez que se han integrado todos los componentes del sistema para crearlo.

protección
Capacidad de un sistema para protegerse contra intrusiones accidentales o premeditadas.

prototipado Mago de Oz
Enfoque para el prototipado de las interfaces de usuario en el que los comandos introducidos por los usuarios son interpretados por una persona quien responde como si fuera la computadora.

reingeniería, proceso de negocio
Cambio de un proceso de negocio para cumplir algún objetivo organizacional nuevo como la reducción de costes y la ejecución más rápida.

reingeniería
Modificación de un sistema software para hacerlo más fácil de comprender y cambiar. La reingeniería a menudo implica la reestructuración y organización de datos y software, la simplificación de programas y la redocumentación.

requerimiento de confiabilidad
Requerimiento del sistema que se incluye para ayudar a alcanzar la confiabilidad requerida para un sistema. Los requerimientos de confiabilidad no funcionales especifican los valores de los atributos de la confiabilidad; los requerimientos de confiabilidad funcionales son requerimientos funcionales para evitar, detectar, tolerar o reponerse de defectos y fallos de funcionamiento del sistema.

requerimiento, funcional
Declaración de alguna función o característica que se debe implementar en un sistema.

requerimiento, no funcional
Declaración de una restricción o comportamiento esperado que se aplica a un sistema. Esta restricción se puede referir a las propiedades emergentes del software que se está desarrollando o al proceso de desarrollo.

riesgo
Resultado indeseable que supone una amenaza para conseguir algún objetivo. Un riesgo del proceso amenaza la agenda o coste de un proceso; un riesgo del producto es un riesgo que puede significar que no se consigan algunos de los requerimientos del sistema.

seguridad
Capacidad de un sistema para funcionar sin fallos de funcionamiento catastróficos.

servicio web
Componente software independiente al que se puede acceder a través de Internet utilizando protocolos estándares. SOAP (Simple Object Access Protocol) se utiliza para el intercambio de información en servicios web. WSDL (Web Service Description Language) se utiliza para definir las interfaces de los servicios web.

servidor
Programa que proporciona algún servicio a otros programas (clientes).

sistema crítico
Sistema informático cuyo fallo de funcionamiento puede causar importantes pérdidas económicas, humanas o medioambientales.

sistema de objetos distribuidos
Sistema distribuido en el que los componentes ejecutables son objetos.

sistema de procesamiento de datos
Sistema cuyo propósito es procesar grandes cantidades de datos estructurados. Estos sistemas normalmente procesan los datos por lotes y siguen un modelo de entrada-proceso-salida. Ejemplos de sistemas de procesamiento de datos son los sistemas de cuentas y facturas, y los sistemas de pago.

sistema de procesamiento de lenguajes
Sistema que traslada un lenguaje a otro. Por ejemplo, un compilador es un sistema de procesamiento de lenguajes que traslada el código fuente de un programa a código objeto.

sistema de procesamiento de transacciones
Sistema que asegura que las transacciones se procesan de tal forma que no se interfieren entre sí y de modo que el fallo de una transacción individual no afecte a otras transacciones o a los datos del sistema.

sistema de tiempo real
Sistema que tiene que responder a eventos externos y procesarlos en «tiempo real». La corrección del sistema no sólo depende de lo que hace sino también de la velocidad con que lo hace. Los sistemas de tiempo real normalmente se organizan como un conjunto de procesos concurrentes que cooperan entre sí.

sistema distribuido
Sistema software en el que los subsistemas o componentes software se ejecutan en diferentes procesadores.

sistema heredado
Sistema socio-técnico que es útil o fundamental para una organización, pero que ha sido desarrollado utilizando una tecnología o métodos obsoletos. Debido a que los sistemas heredados a menudo llevan a cabo funciones de negocio críticas, tienen que ser mantenidos.

sistema peer-to-peer
Sistema distribuido en el que no hay distinción entre clientes y servidores. Las computadoras en el sistema pueden actuar como clientes y como servidores. Entre las aplicaciones peer-to-peer se incluyen la compartición de ficheros, los sistemas de mensajería instantánea y los sistemas de soporte a la cooperación.

sistema socio-técnico
Sistema, incluidos componentes hardware y software, que ha definido los procesos operativos seguidos por los operadores humanos y que funciona dentro de una organización. Por lo tanto, está influidos por las políticas, procedimientos y estructuras de la organización.

sistemas basados en eventos
Sistemas en los que el control del funcionamiento se determina por eventos generados en el entorno del sistema. La mayoría de los sistemas de tiempo real son sistemas basados en eventos.

tipo abstracto de datos
Tipo cuya representación se oculta y está definido por sus operaciones.

tolerancia a defectos
Capacidad de un sistema para continuar en ejecución incluso después de que hayan tenido lugar defectos.

transacción
Unidad de interacción con un sistema informático. Las transacciones son independientes y atómicas (no se pueden dividir en unidades más pequeñas) y son una unidad fundamental de recuperación, consistencia y concurrencia.

validación
Proceso de verificar que un sistema cumple las necesidades y expectativas del cliente.

verificación
Proceso de verificar que un sistema cumple su especificación.

XML
Lenguaje de Marcado Extensible. XML es un lenguaje de marcado de texto que soporta el intercambio de datos estructurados. Cada campo de datos se delimita por etiquetas que proporcionan información sobre ese campo. XML se utiliza ampliamente en la actualidad y se ha convertido en la base de los protocolos para los servicios web.

Z
Lenguaje de especificación formal basado en modelos desarrollado en la Universidad de Oxford en Inglaterra.

Están disponibles definiciones de otros muchos términos en el glosario en línea accesible a través del sitio web del libro.

Abbott, R. (1983). Program design by informal English descriptions. *Comm. ACM*, 26(11), 882–94. (Ch. 14)

Abdel-Ghaly, A. A., Chan, P. Y., *et al.* (1986). Evaluation of competing software reliability predictions. *IEEE Trans. on Software Engineering*, **SE-12**(9), 950–67. (Ch. 24)

Ackroyd, S., Harper, R., *et al.* (1992). *Information Technology and Practical Police Work.* Milton Keynes: Open University Press. (Ch. 2)

Adams, E. N. (1984). Optimizing preventative service of software products. *IBM J. Res. & Dev.*, **28**(1), 2–14. (Ch. 3)

Ahern, D. M., Clouse, A., *et al.* (2001). *CMMI Distilled.* Reading, MA: Addison-Wesley. (Chs. 28, 29)

Albrecht, A. J. (1979). Measuring application development productivity. *Proc. SHARE/GUIDE IBM Application Development Symposium.* (Ch. 26)

Albrecht, A. J. and Gaffney, J. E. (1983). Software function, lines of code and development effort prediction: a software science validation. *IEEE Trans. on Software Engineering*, **SE-9**(6), 639–47. (Ch. 26)

Alexander, C., Ishikawa, S., *et al.* (1977). *A Pattern Language.* Oxford: Oxford University Press. (Ch. 18)

Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling.* New York: John Wiley & Sons. (Ch. 17)

Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems.* Chichester: John Wiley & Sons. (Ch. 24)

Appelrath, H.-J. and Ritter, J. (2000). *SAP R/3 Implementation: Methods and Tools (SAP Excellence).* Berlin: Springer-Verlag. (Ch. 13)

Armour, P. (2002). Ten unmyths of project estimation. *Comm. ACM*, **45**(11), 15–18. (Ch. 26)

Aron, J. D. (1974). *The Program Development Process.* Reading, MA: Addison-Wesley. (Ch. 25)

Arthur, L. J. (1988). *Software Evolution*. New York: John Wiley & Sons. (Ch. 21)

Avizienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Trans. on Software Engineering*, **SE-11**(12), 1491–501. (Ch. 20)

Avizienis, A. A. (1995). A methodology of N-version programming. In *Software Fault Tolerance* (M. R. Lyu, ed.). New York: John Wiley & Sons, 23–46. (Ch. 20)

Bagert, D. J. (2002). Texas licensing of software engineers: all's quiet for now. *Comm. ACM*, **45**(11), 92–4. (Ch. 24)

Baker, F. T. (1972). Chief programmer team management of production programming. *IBM Systems J.*, **11**(1), 56–73. (Ch. 25)

Baker, T. (2002). Lessons learned integrating COTS into systems. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, FL: Springer-Verlag. (Ch. 18)

Balk, L. D. and Kedia, A. (2000). PPT: a COTS integration case study. *Proc. Int. Conf. on Software Engineering*, Limerick, Ireland: ACM Press. (Ch. 18)

Bamford, R. and Deibler, W. J., eds. (2003). ISO 9001: 2000 for Software and Systems Providers: An Engineering Approach. CRC Press. (Ch. 27)

Banker, R. D., Datar, S. M., *et al.* (1993). Software complexity and maintenance costs. *Comm. ACM*, **36**(11), 81–94. (Chs. 21, 26)

Banker, R., Kauffman, R., *et al.* (1994). An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment. *J. of Management Info. Sys.*, **8**(3), 127–50. (Ch. 26)

Bansler, J. P. and Bødker, K. (1993). A reappraisal of structured analysis: design in an organizational context. *ACM Trans. on Information Systems*, **11**(2), 165–93. (Ch. 4)

Barker, R. (1989). *CASE* Method: Entity Relationship Modelling*. Wokingham: Addison-Wesley. (Ch. 8)

Barnard, J. and Price, A. (1994). Managing code inspection information. *IEEE Software*, **11**(2), 59–69. (Chs. 22, 27)

Basili, V. and Green, S. (1993). Software process improvement at the SEL. *IEEE Software*, **11**(4), 58–66. (Ch. 28)

Basili, V. R. and Rombach, H. D. (1988). The TAME project: towards improvement-oriented software environments. *IEEE Trans. on Software Engineering*, **14**(6), 758–73. (Chs. 27, 28)

Bass, B. M. and Dunteman, G. (1963). Behaviour in groups as a function of self, interaction and task orientation. *J. Abnorm. Soc. Psychology*, **66**(4), 19–28. (Ch. 25)

Bass, L., Clements, P., *et al.* (2003). *Software Architecture in Practice, 2nd edn*. Boston: Addison-Wesley. (Ch. 11)

Baumer, D., Gryczan, G., *et al.* (1997). Framework development for large systems. *Comm. ACM*, **40**(10), 52–9. (Ch. 18)

Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer*, **32**(10), 70–8. (Chs. 6, 17)

Beck, K. (2000). *Extreme Programming Explained*. Boston: Addison-Wesley. (Chs. 4, 17, 25, 26)

Beck, K. and Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. *Proc. OOPSLA'89*, New Orleans: ACM Press. (Ch. 14)

Bentley, R., Rodden, T., *et al.* (1992). Ethnographically informed systems design for air traffic control. *Proc. CSCW'92*, Toronto: ACM Press. (Ch. 16)

Berczuk, S. P. and Appleton, B. (2002). *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley. (Ch. 18)

Berghel, H. (2001). The code red worm. *Comm. ACM*, **44**(12), 15–19. (Ch. 3)

Berliner, B. (1990). CVS II: parallelizing software development. *Proc. 1990 Winter USENIX Conference*, Washington, DC: USENIX Assoc. (Ch. 29)

Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Comm. ACM*, **39**(2), 86–97. (Ch. 12)

Bersoff, E. H. and Davis, A. M. (1991). Impact of life cycle models on software configuration management. *Comm. ACM*, **34**(8), 104–18. (Ch. 29)

Bezier, B. (1990). *Software Testing Techniques, 2nd edn.* New York: Van Nostrand Rheinhold. (Ch. 23)

Biggerstaff, T. (1998). A perspective of generative reuse. *Annals of Software Engineering*, **5**, 169–226. (Ch. 18)

Bishop, P. and Bloomfield, R. E. (1995). The SHIP safety case approach. *Proc. Safecomp'95*, Belgirate, Italy: Springer-Verlag. (Ch. 24)

Bishop, P. and Bloomfield, R. E. (1998). A methodology for safety case development. *Proc. Safety-critical Systems Symposium*, Birmingham, UK: Springer-Verlag. (Ch. 24)

Blevins, D. (2001). Overview of the Enterprise Java Beans component model. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 589–606 (Ch. 19)

Boehm, B. (1997). *COCOMO II Model Definition Manual*. Center for Software Engineering, Univ. of Southern California (http://sunset.usc.edu/research/COCOMOII). (Ch. 26)

Boehm, B. and Abts, C. (1999). COTS integration: plug and pray? *IEEE Computer*, **32**(1), 135–8. (Ch. 18)

Boehm, B. and Royce, W. (1989). Ada COCOMO and the Ada process model. *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh: Software Engineering Institute. (Ch. 26)

Boehm, B. W. (1979). Software engineering; R & D trends and defense needs. In *Research Directions in Software Technology* (P. Wegner, ed.). Cambridge, MA: MIT Press, 1–9. (Ch. 22)

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 26)

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, **21**(5), 61–72. (Chs. 4, 5)

Boehm, B. W., Abts, C., *et al.* (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall. (Chs. 19, 22, 26)

Boehm, B. W., McClean, R. L., *et al.* (1975). Some experience with automated aids to the design of large-scale reliable software. *IEEE Trans. on Software Engineering*, **SE-1**(1), 125–33. (Ch. 3)

Boehm, B., Clark, B., *et al.* (1995). Cost models for future life cycle processes: COCOMO 2. *Annals of Software Engineering*, **1**, 57–94. (Ch. 26)

Bolognesi, T. and Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks*, **14**(1), 25–59. (Ch. 10)

Booch, G. (1987). *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, CA: Benjamin-Cummings. (Ch. 18)

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin-Cummings. (Chs. 1, 4, 8, 14)

Booch, G., Rumbaugh, J., *et al.* (1999). *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley. (Ch. 1, 4, 8)

Borchers, J. (2001). *A Pattern Approach to Interaction Design*. New York: John Wiley & Sons. (Ch. 18)

Bosch, J. (2000). *Design and Use of Software Architectures*. Harlow: Addison-Wesley. (Ch. 11)

Bourne, S. R. (1978). The Unix shell. *Bell Sys. Tech. J.*, **57**(6), 1971–90. (Ch. 17)

Bracket, M. H. (1994). *Data Sharing Using a Common Data Architecture*. New York: John Wiley & Sons. (Ch. 13)

Brazendale, J. and Bell, R. (1994). Safety-related control and protection systems: standards update. *IEEE Computing and Control Engineering J.*, **5**(1), 6–12. (Ch. 9)

Brilliant, S. S., Knight, J. C., *et al.* (1990). Analysis of faults in an N-version software experiment. *IEEE Trans. on Software Engineering*, **16**(2), 238–47. (Ch. 20)

Brinch-Hansen, P. (1973). *Operating System Principles*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 15)

Brooks F. P. (1975). *The Mythical Man Month*. Reading, MA: Addison-Wesley. (Ch. 25)

Brown, A. W., Earl, A. N., *et al.* (1992). *Software Engineering Environments*. London: McGraw-Hill. (Ch. 11)

Budgen, D. (2003). *Software Design, 2nd edn*. Harlow: Addison-Wesley. (Ch. 8)

Burns, A. and Wellings, A. (2001). *Real-Time Systems and Programming Languages*. Harlow: Addison-Wesley. (Ch. 15)

Butler, H. (1994). Virtual remote: the centralized expert. *HP Journal*, **45**(5), http://www.hpl.hp.com/hpjournal/94oct/oct94a13.htm. (Ch. 18)

Buxton, J. (1980). *Requirements for Ada Programming Support Environments: Stoneman*. Washington, DC: US Department of Defense. (Ch. 11)

Checkland, P. (1981). *Systems Thinking, Systems Practice*. Chichester: John Wiley & Sons. (Ch. 2)

Checkland, P. and Scholes, J. (1990). *Soft Systems Methodology in Action*. Chichester: John Wiley & Sons. (Ch. 2)

Chen, P. (1976). The entity relationship model—towards a unified view of data. *ACM Trans. on Database Systems*, **1**(1), 9–36. (Ch. 8)

Chidamber, S. and Kemerer, C. (1994). A metrics suite for object-oriented design. *IEEE Trans. on Software Engineering*, **20**(6), 476–93. (Ch. 27)

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: a taxonomy. *IEEE Software*, **7**(1), 13–17. (Ch. 21)

Clements, P., Bachmann, F., *et al.* (2002). *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley. (Ch. 11)

Coad, P. and Yourdon, E. (1990). *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice Hall. (Chs. 8, 14)

Cobb, R. H. and Mills, H. D. (1990). Engineering software under statistical quality control. *IEEE Software*, **7**(6), 44–54. (Ch. 22)

Cockburn, A. (2001). *Agile Software Development*. Reading, MA: Addison-Wesley. (Ch. 17)

Codd, E. F. (1979). Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, **4**(4), 397–434. (Ch. 8)

Cohen, B., Harwood, W. T., *et al.* (1986). *The Specification of Complex Systems*. Wokingham: Addison-Wesley. (Ch. 10)

Constantine, L. L. and Yourdon, E. (1979). *Structured Design*. Englewood Cliffs, NJ: Prentice Hall. (Chs. 4, 8)

Cooling, J. (2003). *Software Engineering for Real-Time Systems*. Harlow: Addison-Wesley. (Ch. 15)

Coulouris, G., Dollimore, J., *et al.* (2001). *Distributed Systems: Concepts and Design*. Harlow: Addison-Wesley. (Ch. 12)

Councill, W. T. and Heineman, G. T. (2001). Definition of a software component and its elements. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 5–20. (Ch. 19)

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography.* London: Springer. (Ch. 16)

Crosby, P. (1979). *Quality Is Free.* New York: McGraw-Hill. (Ch. 27)

Curtis, B., Hefley, W. E., *et al.* (2001). *The People Capability Model: Guidelines for Improving the Workforce.* Boston: Addison-Wesley. (Chs. 25, 28)

Cusamano, M. (1989). The software factory: a historical interpretation. *IEEE Software,* **6**(2), 23–30. (Ch. 18)

Czarnecki, K. and Eisenecher, U. (2000). *Generative Programming: Methods, Tools, and Applications.* Boston: Addison-Wesley. (Ch. 18)

Davis, A. M. (1993). *Software Requirements: Objects, Functions, & States.* Englewood Cliffs, NJ: Prentice Hall. (Ch. 6)

Dehbonei, B. and Mejia, F. (1995). Formal development of safety-critical software systems in railway signalling. In *Applications of Formal Methods* (M. Hinchey and J. P. Bowen, eds.). London: Prentice Hall, 227–52. (Ch. 10)

DeMarco, T. (1978). *Structured Analysis and System Specification.* New York: Yourdon Press. (Ch. 8)

DeMarco, T. and Boehm, B. (2002). The agile methods fray. *IEEE Computer,* **35**(6), 90–2. (Ch. 17)

DeMarco, T. and Lister, T. (1985). Programmer performance and the effects of the workplace. *Proc. 8th Int. Conf. on Software Engineering,* London: IEEE Press. (Ch. 25)

DeMarco, T. and Lister, T. (1999). *Peopleware: Productive Projects and Teams.* New York: Dorset House. (Ch. 25)

DeMarco, T. (1978). *Structured Analysis and System Specification.* New York: Yourdon Press. (Ch. 1)

Diaper, D. (1989). *Task Analysis for Human-Computer Interaction.* Chichester: Ellis Horwood. (Ch. 16)

Dijkstra, E. W. (1968). Cooperating sequential processes. In *Programming Languages* (F. Genuys, ed.). London: Academic Press, 43–112. (Ch. 15)

Dijkstra, E. W. (1968). Goto statement considered harmful. *Comm. ACM,* **11**(3), 147–8. (Ch. 20)

Dijkstra ,E. W., Dahl, O. J., *et al.* (1972). *Structured Programming.* London: Academic Press. (Ch. 23)

Dix, A., Finlay, J., *et al.* (2004). *Human Computer Interaction, 3rd edn.* Harlow: Addison-Wesley. (Ch. 16)

Douglass, B. P. (1999). *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd edn.* Boston: Addison-Wesley. (Ch. 15)

DuBois, P. (1996). *Software Portability with imake, 2nd edn.* Sebastopol, CA: O'Reilly & Associates. (Ch. 29)

Easterbrook, S., Lutz, R., *et al.* (1998). Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. on Software Engineering,* **24**(1), 4–14. (Ch. 10)

ECMA. (1991). A reference model for frameworks of computer-assisted software engineering environments. In *Reprints of the Seventh International Software Process Workshop,* Yountville, CA: ACM Press. (Ch. 11)

Ehrlich, W., Prasanna, B., *et al.* (1993). Determining the cost of a stop-test decision. *IEEE Software,* **9**(4), 33–42. (Ch. 24)

El-Amam, K. (2001). *Object-Oriented Metrics: A Review of Theory and Practice.* (Ch. 27)

Elliott, J., Eckstein, R., *et al.* (2002). *Java Swing, 2nd edn.* Sebastopol, CA: O'Reilly & Associates Inc. (Ch. 16)

Ellison, R. J., Fisher, D. A., *et al.* (1999). Survivability: protecting your critical systems. *IEEE Internet Computing*, **3**(6), 55–63. (Ch. 3)

Ellison, R. J., Linger, R. C., *et al.* (1999). Survivable network system analysis: a case study. *IEEE Software*, **16**(4), 70–7. (Ch. 3)

Ellison, R., Linger, R., *et al.* (2002). Foundations of survivable systems engineering. *Crosstalk: The Journal of Defense Software Engineering*, **12**, 10–15. (Ch. 3)

Elrad, T., Askit, M., *et al.* (2001). Discussing aspects of AOP. *Comm. ACM*, **44**(10), 33–8. (Ch. 18)

Endres, A. (1975). An analysis of errors and their causes in system programs. *IEEE Trans. on Software Engineering*, **SE-1**(2), 140–9. (Ch. 3)

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Pro*, **May/June 2000**, 17–23. (Ch. 21)

Estublier, J. and Casallas, R. (1994). The Adele configuration manager. In *Configuration Management* (W. Tichy, ed.). Chichester: John Wiley & Sons, 99–134. (Ch. 29)

Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, **19**(1), 42–51. (Ch. 22)

Ewald, T. (2001). Overview of COM+. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 573–88. (Ch. 19)

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems J.*, **15**(3), 182–211. (Ch. 22)

Fagan, M. E. (1986). Advances in software inspections. *IEEE Trans. on Software Engineering*, **SE-12**(7), 744–51. (Ch. 22)

Fayad, M. E. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Comm. ACM*, **40**(10), 32–8. (Ch. 18)

Feldman, S. I. (1979). MAKE—a program for maintaining computer programs. *Software-Practice and Experience*, **9**(4), 255–65. (Ch. 29)

Firesmith, D. G. (2003). Analysing and specifying reusable security requirements. *Journal of Object Technology*, **2**(1), 53–68. (Ch. 9)

Foster, I., Kesselman, C., *et al.* (2002). Grid services for distributed system integration. *IEEE Computer*, **35**(6), 37–46. (Ch. 12)

Frewin, G. D. and Hatton, B. J. (1986). Quality management—procedures and practises. *IEE/BCS Software Engineering J.*, **1**(1), 29–38. (Ch. 22)

Fromme, B. and Walker, J. (1993). An open architecture for tool and process integration. *Proc. 6th Conf. on Software Engineering Environments*, Reading, UK: IEEE Press. (Ch. 11)

Fuggetta, A. (1993). A classification of CASE technology. *IEEE Computer*, **26**(12), 25–38. (Ch. 4)

Fujiwara, E. and Pradhan, D. K. (1990). Error-control coding in computers. *IEEE Computer*, **23**(7), 63–72. (Ch. 20)

Furey, S. and Kitchenham, B. (1997). Point/counterpoint: function points. *IEEE Software*, **14**(2), 28–31. (Ch. 26)

Futatsugi, K., Goguen, J. A., *et al.* (1985). Principles of OBJ2. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans: ACM Press. (Ch. 10)

Gamma, E., Helm, R., *et al.* (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley. (Ch. 18)

Gane, C. and Sarson, T. (1979). *Structured Systems Analysis*. Englewood Cliffs, NJ: Prentice Hall. (Chs. 4, 8)

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, **1**, 1–39. (Chs. 11, 13)

Garlan, D., Allen, R., *et al.* (1995). Architectural mismatch: why reuse is so hard. *IEEE Software*, **12**(6), 17–26. (Ch. 18)

Garlan, D., Kaiser, G. E., *et al.* (1992). Using tool abstraction to compose systems. *IEEE Computer*, **25**(6), 30–8. (Ch. 11)

Garmus, D. and Herron, D. (2000). *Function Point Analysis: Measurement Practices for Successful Software Projects*. Boston: Addison-Wesley. (Ch. 26)

Gilb, T. and Graham, D. (1993). *Software Inspection*. Wokingham: Addison-Wesley. (Ch. 22)

Goldberg, A. and Robson, D. (1983). *Smalltalk-80. The Language and Its Implementation*. Reading, MA: Addison-Wesley. (Ch. 16)

Gollmann, D. (1999). *Computer Security*. Chichester: John Wiley & Sons. (Ch. 24)

Gomaa, H. (1993). *Software Design Methods for Concurrent and Real-Time Systems*. Reading, MA: Addison-Wesley. (Ch. 15)

Gordon, V. S. and Bieman, J. M. (1995). Rapid prototyping: lessons learned. *IEEE Software*, **12**(1), 85–95. (Ch. 17)

Gotterbarn, D., Miller, K., et al. (1999). Software engineering code of ethics is approved. *Comm. ACM*, **42**(10), 102–7. (Ch. 1)

Grady, R. B. (1993). Practical results from measuring software quality. *Comm. ACM*, **36**(11), 62–8. (Ch. 27)

Grady, R. B. and Van Slack, T. (1994). Key lessons in achieving widespread inspection use. *IEEE Software*, **11**(4), 46–57. (Ch. 22)

Graham, I. (1994). *Object-Oriented Methods, 2nd edn*. Wokingham: Addison-Wesley. (Ch. 14)

Griss, M. L. and Wosser, M. (1995). Making reuse work at Hewlett-Packard. *IEEE Software*, **12**(1), 105–7. (Ch. 18)

Groff, J. R., Weinberg, P. N., *et al.* (2002). *SQL: The Complete Reference, 2nd edn*. New York: McGraw-Hill Osborne. (Ch. 17)

Grudin, J. (1989). The case against user interface consistency. *Comm. ACM*, **32**(10), 1164–73. (Ch. 16)

Guimaraes, T. (1983). Managing application program maintenance expenditures. *Comm. ACM*, **26**(10), 739–46. (Ch. 21)

Gunning, R. (1962). *Techniques of Clear Writing*. New York: McGraw-Hill. (Ch. 27)

Guttag, J. (1977). Abstract data types and the development of data structures. *Comm. ACM*, **20**(6), 396–405. (Ch. 10)

Guttag, J., Horning, J., *et al.* (1993). *Larch: Languages and Tools for Formal Specification*. Heidelberg: Springer-Verlag. (Ch. 10)

Haase, V., Messnarz, R., *et al.* (1994). Bootstrap: fine tuning process assessment. *IEEE Software*, **11**(4), 25–35. (Ch. 28)

Hall, A. (1990). Seven myths of formal methods. *IEEE Software*, **7**(5), 11–20. (Ch. 10)

Hall, A. (1996). Using formal methods to develop an ATC information system. *IEEE Software*, **13**(2), 66–76. (Chs. 3, 9, 10)

Hall, A. and Chapman, R. (2002). Correctness by construction: developing a commercially secure system. *IEEE Software*, **19**(1), 18–25. (Ch. 3, 9, 10)

Hall, E. (1998). *Managing Risk: Methods for Software Systems Development*. Reading, MA: Addison-Wesley. (Ch. 5)

Hall, T. and Fenton, N. (1997). Implementing effective software metrics programs. *IEEE Software*, **14**(2), 55–64. (Ch. 27)

Halstead, M. H. (1977). *Elements of Software Science*. Amsterdam: North-Holland. (Ch. 21)

Hamlet, D. (1992). Are we testing for true reliability? *IEEE Software*, **9**(4), 21–7. (Ch. 24)

Hammer, M. (1990). Reengineering work: don't automate, obliterate. *Harvard Business Review*, **July-August 1990**, 104–12. (Ch. 28)

Hammer, M. and McLeod, D. (1981). Database descriptions with SDM: A semantic database model. *ACM Trans. on Database Sys.*, **6**(3), 351–86. (Ch. 8)

Hardin, D., Frerking, M., *et al.* (2002). Getting down and dirty: device-level programming using the real-time specification for Java. *Proc. Fifth IEEE International Symp. on Object-Oriented Real-Time Distributed Computing*, Washington, DC: IEEE Computer Society Press. (Ch. 15)

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Sci. Comput. Programming*, **8**(3), 231–74. (Chs. 8, 14, 15)

Harel, D. (1988). On visual formalisms. *Comm. ACM*, **31**(5), 514–30. (Chs. 8, 15)

Harold, E. R. and Means, W. S. (2002). *XML in a Nutshell*. Sebastopol. CA: O'Reilly & Associates. (Ch. 13)

Hass, A. M. J. (2003). *Configuration Management: Principles and Practice*. Boston: Addison-Wesley. (Ch. 29)

Hayes, I. (1987). *Specification Case Studies*. London: Prentice Hall. (Ch. 10)

Heninger, K. L. (1980). Specifying software requirements for complex systems: new techniques and their applications. *IEEE Trans. on Software Engineering*, **SE-6**(1), 2–13. (Ch. 6)

Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House. (Ch. 17)

Higuera-Toledano, M. T. and Issarny, V. (2000). Java embedded real-time systems: an overview of existing solutions. *Proc. Third IEEE International Symp. on Object-Oriented Real-Time Distributed Computing*, Newport Beach, CA: IEEE Computer Society Press. (Ch. 15)

Hoare, C. A. R. (1974). Monitors: an operating system structuring concept. *Comm. ACM*, **21**(8), 666–77. (Ch. 15)

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. London: Prentice Hall. (Ch. 10)

Hofmeister, C., Nord, R., *et al.* (2000). *Applied Software Architecture*. Boston: Addison-Wesley. (Ch. 11)

Horswill, J. and Miller, S. A. (2000). *Designing and Programming CICS Applications*. Sebastopol, CA: O'Reilly & Associates. (Ch. 13)

Huang, Y. and Kintala, C. M. R. (1993). Software implemented fault tolerance: technologies and experience. *Proc. 23rd Fault-tolerant Computing Symposium (FTCS-23)*, Toulouse, France: IEEE Computer Society Press. (Ch. 20)

Huff, C. C. (1992). Elements of a realistic CASE tool adoption budget. *Comm. ACM*, **35**(4), 45–54. (Ch. 4)

Huff, K. E. (1996). Software process modeling. In *Trends in Software: Software Process* (A. Fuggetta and A. Wolf, eds.). Chichester: John Wiley & Sons, 1–24. (Ch. 28)

Huff, C. and Martin, C. D. (1995). Computing consequences: a framework for teaching ethical computing. *Comm. ACM*, **38**(12), 75–84. (Ch. 1)

Hughes, J. A., O'Brien, J., *et al.* (1997). Designing with ethnography: a presentation framework for design. *Proc. DIS'97*, Amsterdam: ACM Press. (Ch. 16)

Hull, R. and King, R. (1987). Semantic database modeling: survey, applications and research issues. *ACM Computing Surveys*, **19**(3), 201–60. (Ch. 8)

Humphrey, W. (1989). *Managing the Software Process*. Reading, MA: Addison-Wesley. (Chs. 22, 27. 28)

Humphrey, W. S. (1988). Characterizing the software process. *IEEE Software*, **5**(2), 73–9. (Ch. 28)

Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley. (Ch. 28)

IEC. (1998). *Standard IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems*. (Ch. 9)

IEEE. (1998). IEEE recommended practice for software requirements specifications. In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press. (Ch. 6)

IEEE. (2003). *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, CA: IEEE Computer Society Press. (Ch. 27)

Ince, D. (1994). *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill. (Ch. 27)

Jackson, M. A. (1983). *System Development*. London: Prentice Hall. (Chs. 1, 8)

Jackson, M. A. (1995). *Requirements and Specifications*. Wokingham: Addison-Wesley. (Ch. 6)

Jacky, J. (1995). Specifying a safety-critical control system. *IEEE Trans. on Software Engineering*, **21**(2), 99–106. (Ch. 10)

Jacky, J. (1997). *The Way of Z: Practical Programming with Formal methods*. Cambridge, UK: Cambridge University Press. (Ch. 10)

Jacky, J., Unger, J., *et al.* (1997). Experience with Z: developing a control program for a radiation therapy machine. *Proc. ZUM'97*, Reading: Springer. (Ch. 10)

Jacobsen, I., Christerson, M., *et al.* (1993). *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley. (Chs. 6, 8, 15)

Jacobsen, I., Griss, M., *et al.* (1997). *Software Reuse*. Reading, MA: Addison-Wesley. (Chs. 18, 19)

Jahanian, F. and Mok, A. K. (1986). Safety analysis of timing properties in real-time systems. *IEEE Trans. on Software Engineering*, **SE-12**(9), 890–904. (Ch. 9)

Janis, I. L. (1972). *Victims of Groupthink. A Psychological Study of Foreign Policy Decisions and Fiascos*. Boston: Houghton Mifflin. (Ch. 25)

Jelinski, Z. and Moranda, P. B. (1972). Software reliability research. In *Statistical Computer Performance Evaluation* (W. Frieberger, ed.). New York: Academic Press, 465–84. (Ch. 24)

Johnson, P. L. (1993). *ISO 9000: Meeting the New International Standards*. New York: McGraw-Hill. (Ch. 27)

Jones, C. B. (1980). *Software Development—A Rigorous Approach*. London: Prentice Hall. (Ch. 10)

Jones, C. B. (1986). *Systematic Software Development Using VDM*. London: Prentice Hall. (Chs. 10, 22)

Kafura, D. and Reddy, G. R. (1987). The use of software complexity metrics in software maintenance. *IEEE Trans. on Software Engineering*, **SE-13**(3), 335–43. (Ch. 21)

Kan, S. H. (2003). *Metrics and Models in Software Quality Engineering*. Boston: Addison-Wesley. (Ch. 24)

Kiczales, G., Hilsdale, E., *et al.* (2001). Getting started with AspectJ. *Comm. ACM*, **44**(10), 59–65. (Ch. 18)

Kilpi, T. (2001). Implementing a software metrics program at Nokia. *IEEE Software*, **18**(6), 72–7. (Ch. 27)

Kit, E. (1995). *Software Testing in the Real World: Improving the Process*. Reading, MA: Addison-Wesley. (Ch. 22)

Kitchenham, B. (1990). Measuring software development. In *Software Reliability Handbook* (P. Rook, ed.). Amsterdam: Elsevier, 303–31. (Ch. 27)

Kleppe, A., Warmer, J., *et al.* (2003). *MDA Explained: The Model-Driven Architecture—Practice and Promise*. Boston: Addison-Wesley. (Ch. 14)

Knight, J. C. and Leveson, N. G. (1986). An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Software Engineering*, **SE-12**(1), 96–109. (Ch. 20)

Knight, J. C. and Leveson, N. G. (2002). Should software engineers be licensed? *Comm. ACM*, **45**(11), 87–90. (Ch. 24)

Knuth, D. E. (1971). *The Art of Computer Programming: Fundamental Algorithms*. Reading, MA: Addison-Wesley. (Ch. 18)

Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. Chichester: John Wiley & Sons. (Ch. 6)

Kreger, H. (2001). *Web Services Conceptual Architecture (WSCA 1.0)*. IBM. www.ibm.com/software/solutions/ webservices/pdf/WSCA.pdf (Ch. 12)

Krutchen, P. (2000). *The Rational Unified Process—An Introduction*. Reading, MA: Addison-Wesley. (Chs. 4, 8)

Kumaran, S. I. (2001). *JINI Technology: An Overview*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 12)

Kuvaja, P., Similä, J., *et al.* (1994). *Software Process Assessment and Improvement: The BOOTSTRAP Approach*. Oxford: Blackwell Publishers. (Ch. 28)

Lamping, J., Rao, R., *et al.* (1995). A focus + context technique based on hyperbolic geometry for visualising large hierarchies. *Proc. CHI'95*, Denver, CO: ACM Press. (Ch. 16)

Laprie, J.-C. (1995). Dependable computing: concepts, limits, challenges. *Proc. 25th IEEE Symposium on Fault-Tolerant Computing*, Pasadena, CA: IEEE Press. (Ch. 3)

Laprie, J.-C., Arlat, J., *et al.* (1995). Architectural issues in software fault tolerance. In *Software Fault Tolerance* (M. R. Lyu, ed.). New York: John Wiley & Sons 47–80. (Ch. 20)

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 17)

Laudon, K. (1995). Ethical concepts and information technology. *Comm. ACM*, **38**(12), 33–9. (Ch. 1)

Leblang, D. B. and Chase, R. P. (1987). Parallel software configuration management in a network environment. *IEEE Software*, **4**(6), 28–35. (Ch. 29)

Lehman, M. M. (1996). Laws of software evolution revisited. *Proc. European Workshop on Software Process Technology (EWSPT'96)*, Nancy, France: Springer-Verlag. (Ch. 21)

Lehman, M. M. and Belady, L. (1985). *Program Evolution: Processes of Software Change*. London: Academic Press. (Ch. 21)

Lehman, M. M., Perry, D. E., *et al.* (1998). On evidence supporting the FEAST hypothesis and the laws of software evolution. *Proc. Metrics'98*, Bethesda, MD: IEEE Computer Society Press. (Ch. 21)

Lehman, M. M., Ramil, J. F., *et al.* (2001). An approach to modelling long-term growth trends in software systems. *Proc. Int. Conf. on Software Maintenance*, Florence, Italy: IEEE Computer Society Press. (Ch. 21)

Leveson, N. and Stolzy, J. (1987). Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, **13**(3), 386–97. (Ch. 9)

Leveson, N. G. (1985). Software safety. In *Resilient Computing Systems* (T. Anderson, ed.). London: Collins, 12343. (Chs. 3, 9)

Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley. (Chs. 9, 20)

Leveson, N. G. and Harvey, P. R. (1983). Analysing software safety. *IEEE Trans. on Software Engineering*, **SE-9**(5), 569–79. (Ch. 9)

Lewis, P. M., Bernstein, A. J., *et al.* (2003). *Databases and Transaction Processing: An Application-Oriented Approach*. Boston: Addison-Wesley. (Ch. 13)

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Reading, MA: Addison-Wesley. (Ch. 21)

Linger, R. C. (1994). Cleanroom process model. *IEEE Software*, **11**(2), 50–8. (Chs. 4, 22)

Liskov, B. and Guttag, J. (1986). *Abstraction and Specification in Program Development*. Cambridge, MA: MIT Press. (Ch. 10)

Littlewood, B. (1990). Software reliability growth models. In *Software Reliability Handbook* (P. Rook, ed.). Amsterdam: Elsevier, 401–12. (Chs. 3, 24)

Littlewood, B. and Verrall, J. L. (1973). A Bayesian reliability growth model for computer software. *Applied Statistics*, **22**, 332–46. (Ch. 24)

Londeix, B. (1987). *Cost Estimation for Software Development*. Wokingham: Addison-Wesley. (Ch. 26)

Lovelock, C., Vandermerwe, S., *et al.* (1996). *Services Marketing*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 12)

Lutz, M. (1996). *Programming Python*. Sebastopol, CA: O'Reilly & Associates. (Ch. 17)

Lutz, R. R. (1993). Analysing software requirements errors in safety-critical embedded systems. *Proc. RE'93*, San Diego CA: IEEE Computer Society Press. (Chs. 3, 22, 23)

MacDonell, S. G. (1994). Comparative review of functional complexity assessment methods for effort estimation. *BCS/IEE Software Engineering J.*, **9**(3), 107–17. (Ch. 26)

Marshall, J. E. and Heslin, R. (1975). Boys and girls together: sexual composition and the effect of density on group size and cohesiveness. *J. of Personality and Social Psychology*, **35**(5), 952–61. (Ch. 25)

Martin, D., Rodden, T., *et al.* (2001). Finding patterns in the fieldwork. *Proc. ECSCW'01*, Bonn: Kluwer. (Ch. 18)

Martin, D., Rouncefield, M., *et al.* (2002). Applying patterns of interaction to work (re)design: e-government and planning. *Proc CHI'2002*, ACM Press. (Ch. 18)

Maslow, A. A. (1954). *Motivation and Personality*. New York: Harper and Row. (Ch. 25)

Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, CT: Manning. (Ch. 23)

Matsumoto, Y. (1984). Some experience in promoting reusable software: presentation in higher abstract levels. *IEEE Trans. on Software Engineering*, **SE-10**(5), 502–12. (Ch. 18)

McCabe, T. J. (1976). A complexity measure. *IEEE Trans. on Software Engineering*, **SE-2**(4), 308–20. (Ch. 21)

McCue, G. M. (1978). IBM's Santa Teresa laboratory: architectural design for program development. *IBM Systems J.*, **17**(1), 4–25. (Ch. 25)

McDougall, P. (2000). The power of peer-to-peer. *Information Week*, August 28, http://www.informationweek.com. (Ch. 12)

McGuffin, R. W., Elliston, A. E., *et al.* (1979). CADES—software engineering in practice. *Proc. 4th Int. Conf. on Software Engineering*, Munich: IEEE Computer Society Press. (Ch. 11)

McIlroy, M. D. (1968). Mass-produced software components. *Proc. NATO Conf. on Software Engineering*, Garmisch, Germany: Springer-Verlag. (Ch. 18)

Meyer, B. (1992). Design by contract. *IEEE Computer*, **25**(10), 40–51. (Ch. 19)

Meyer, B. (2003). The grand challenge of trusted components. *Proc. ICSE 25: Int. Conf. on Software Engineering*, Portland, OR: IEEE Press. (Ch. 19)

Mili, H., Mili, A., *et al.* (2002). *Reuse-Based Software Engineering*. New York: John Wiley & Sons. (Ch. 19)

Miller, G. A. (1957). The magical number 7 plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, **63**, 81–97. (Ch. 16)

Mills, H. D., Dyer, M., *et al.* (1987). Cleanroom software engineering. *IEEE Software*, **4**(5), 19–25. (Chs. 3, 4, 22)

Mitschele-Thiel, A. (2001). *Systems Engineering with SDL: Developing Performance-Critical Communication Systems*. Chichester: John Wiley & Sons. (Ch. 22)

MOD, (1995). *The Procurement of Safety Critical Software (Revised edn)*. UK Ministry of Defence, Interim Standard, 00–55. (Ch. 22)

Mosley, D. J. and Posey, B. A. (2002). *Just Enough Test Automation*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 23)

Mumford, E. (1989). User participation in a changing environment—why we need it. In *Participation in Systems Development* (K. Knight, ed.). London: Kegan Paul. (Ch. 2)

Munch, B. P., Larsen, J-O., *et al.* (1993). Uniform versioning: the change-oriented model. *Proc. 4th Workshop on Software Configuration Management*, Baltimore, MD: ACM Press. (Ch. 29)

Musa, J. D. (1993). Operational profiles in software reliability engineering. *IEEE Software*, **10**(2), 14–32. (Ch. 24)

Musa, J. D. (1998). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill. (Ch. 24)

Musciano, C. and Kennedy, B. (2002). *HTML & XHTML: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates. (Ch. 16)

Myers, W. (1989). Allow plenty of time for large-scale software. *IEEE Software*, **6** (4), 92–9. (Ch. 26)

Nakajo, T. and Kume, H. (1991). A case history analysis of software error-cause relationships. *IEEE Trans. on Software Engineering*, **18**(8), 830–8. (Ch. 3)

Neil, M., Ostrolenk, G., *et al.* (1998). Lessons from using Z to specify a software tool. *IEEE Trans. on Software Engineering*, **24**(1), 15–23. (Ch. 10)

Neilsen ,J. (1993). *Usability Engineering*. New York: Academic Press. (Ch. 16)

Nii, H. P. (1986). Blackboard systems, parts 1 and 2. *AI Magazine*, 7(3 and 4), 38–53 and 62–9. (Ch. 11)

Nilsen, K. (1998). Adding real-time capabilities to Java. *Comm. ACM*, **41**(6), 49–56. (Ch. 15)

Norman, D. A. and Draper, S. W. (1986). *User-Centered System Design*. Hillsdale, NJ: Lawrence Erlbaum. (Ch. 16)

Nosek, J. T. and Palvia, P. (1990). Software maintenance management: changes in the last decade. *Software Maintenance: Research and Practice*, **2**(3), 157–74. (Ch. 21)

Nuseibeh, B. (1997). Ariane 5: who dunnit? *IEEE Software*, **14**(3), 15–16. (Ch. 18)

O'Connor, J., Mansour, C., *et al.* (1994). Reuse in command and control systems. *IEEE Software*, **11**(4), 70–9. (Ch. 18)

Offen, R. J. and Jeffrey, R. (1997). Establishing software measurement programs. *IEEE Software*, **14**(2), 45–54. (Ch. 27)

O'Leary, D. E. (2000). *Enterprise Resource Planning Systems: Systems, Life Cycle, Electronic Commerce and Risk*. Cambridge, UK: Cambridge University Press. (Ch. 18)

Oram, A. (2001). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates. (Ch. 12)

Oram, A. and Talbott, S. (1991). *Managing Projects with make, 2nd edn*. Sebastopol, CA: O'Reilly & Associates. (Ch. 29)

Orcero, D. S. (2000). The code analyser LCLint. *Linux Journal*, 73, http://www.linuxjournal.com/article.php?sid=3599. (Ch. 22)

Orfali, R. and Harkey, D. (1998). *Client/Server Programming with Java and CORBA*. New York: John Wiley & Sons. (Ch. 12)

Oskarsson, O. and Glass, R. L. (1995). *An ISO 9000 Approach to Building Quality Software*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 27)

Ould, M. A. (1999). *Managing Software Quality and Business Risk*. Chichester: John Wiley & Sons. (Ch. 5)

Ould, M. A. (1995). *Business Processes: Modelling and Analysis for Re-engineering and Improvement*. Chichester: John Wiley & Sons. (Ch. 28)

Ousterhout, J. (1994). *TCL and the TK toolkit*. Reading, MA: Addison-Wesley. (Ch. 17)

Ousterhout, J. K. (1998). Scripting: higher-level programming for the 21st century. *IEEE Computer*, **31**(3), 23–30. (Chs. 17, 26)

Palmer, S. R. and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 17)

Parnas, D. L., van Schouwen, J., *et al.* (1990). Evaluation of safety-critical software. *Comm. ACM*, **33** (6), 636–51. (Chs. 20, 24)

Paulk, M. C. and Konrad, M. (1994). An overview of ISO's SPICE project. *IEEE Computer*, **27**(4), 68–70. (Ch. 28)

Paulk, M. C., Curtis, B., *et al.* (1993). Capability maturity model, version 1.1. *IEEE Software*, **10**(4), 18–27. (Ch. 28)

Paulk, M. C., Weber, C. V., *et al.* (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley. (Chs. 28, 29)

Peach, R. W. (1996). *The ISO 9000 Handbook, 3rd edn*. New York: Irwin Professional. (Chs. 27, 29)

Perrow, C. (1984). *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books. (Ch. 3)

Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. New York: McGraw-Hill. (Chs. 9, 10)

Pfaff, G. and ten Hagen, P. J. W. (1985). *Seeheim Workshop on User Interface Management Systems*. Heidelberg: Springer-Verlag. (Ch. 16)

Pfarr, T. and Reis, J. E. (2002). The integration of COTS/GOTS within NASA's HST command and control system. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, FL: Springer-Verlag. (Ch. 18)

Pfleeger, C. P. (1997). *Security in Computing, 2nd edn*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 3)

Pope, A. (1998). *CORBA*. Harlow: Addison-Wesley. (Ch. 19)

Potter, B., Sinclair, J., *et al.* (1996). *An Introduction to Formal Specification and Z*. London: Prentice Hall. (Ch. 10)

Preiser, W. and Ostoff, E. (2001). *The Universal Design Handbook*. New York: McGraw-Hill. (Ch. 16)

Pritchard, J. (1999). *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. Boston: Addison-Wesley. (Ch. 12)

Prowell, S. J., Trammell, C. J., *et al.* (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley. (Chs. 4, 10, 22, 24)

Pulford, K., Kuntzmann-Combelles, A., *et al.* (1996). *A Quantitative Approach to Software Management*. Wokingham: Addison-Wesley. (Chs. 27, 28)

Pullum, L. L. (2001). *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House. (Ch. 20)

Putnam, L. H. (1978). A general empirical solution to the macro software sizing and estimating problem. *IEEE Trans. on Software Engineering*, **SE-4**(3), 345–61. (Ch. 26)

Randell, B. (1975). System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, **SE-1**(2), 220–32. (Ch. 20)

Randell, B. and Xu, J. (1995). The evolution of the recovery block concept. In *Software Fault Tolerance* (M. R. Lyu, ed.). New York: John Wiley & Sons, 1–22. (Ch. 20)

Rankin, C. (2002). The software testing automation framework. *IBM Systems J.*, **41**(1), 126–40. (Ch. 23)

Redmill, F. (1998). IEC 61508: principles and use in the management of safety. *IEEE Computing and Control Engineering J.*, **9**(10), 205–13. (Ch. 9)

Reiss, S., P. (1990). Connecting tools using message passing in the field environment. *IEEE Software*, **7**(4), 57–66. (Ch. 11)

Rettig, M. (1994). Practical programmer: prototyping for tiny fingers. *Comm. ACM*, **37**(4), 21–7. (Ch. 17)

Rittel, H. and Webber, M. (1973). Dilemmas in a general theory of planning. *Policy Sciences*, 4, 155–69. (Ch. 2)

Robertson, S. and Robertson, J. (1999). *Mastering the Requirements Process*. Harlow: Addison-Wesley. (Ch. 6)

Robinson, P. J. (1992). *Hierarchical Object-Oriented Design*. Englewood Cliffs, NJ: Prentice Hall. (Chs. 4, 8, 14)

Ross, D. T. (1977). Structured analysis (SA): a language for communicating ideas. *IEEE Trans. on Software Engineering*, **SE-3**(1), 16–34. (Ch. 6)

Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON*, Los Angeles CA: IEEE Computer Society Press. (Ch. 4)

Rubin, K. and Goldberg, A. (1992). Object behaviour analysis. *Comm. ACM*, **35**(9), 48–62. (Ch. 14)

Rumbaugh, J., Blaha, M., *et al.* (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall. (Chs. 1, 4, 8)

Rumbaugh, J., Jacobson, I., *et al.* (1999). *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley. (Chs. 1, 4, 8, 14)

Rumbaugh, J., Jacobson, I., *et al.* (1999). *The Unified Software Development Process*. Reading, MA: Addison-Wesley. (Chs. 1, 4, 8)

Sackman, H., Erikson, W. J., *et al.* (1968). Exploratory experimentation studies comparing online and off-line programming performance. *Comm. ACM*, **11**(1), 3–11. (Ch. 26)

Schmidt, D. C. (1997). Applying design patterns and frameworks to develop object-oriented communications software. In *Handbook of Programming Languages, Vol. I* (P. Salus, ed.). London: Macmillan Computer Publishing. (Ch. 18)

Schneidewind, N. F. and Keller, T. W. (1992). Applying reliability models to the space shuttle. *IEEE Software*, **9**(4), 28–33. (Ch. 24)

Schoman, K. and Ross, D. T. (1977). Structured analysis for requirements definition. *IEEE Trans. on Software Engineering*, **SE-3**(1), 6–15. (Ch. 6)

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 17)

Scott, J. E. (1999). The FoxMeyer Drug's bankruptcy: was it a failure of ERP? *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*, Milwaukee, WI. (Ch. 18)

Selby, R. W., Basili, V. R., *et al.* (1987). Cleanroom software development: an empirical evaluation. *IEEE Trans. on Software Engineering*, **SE-13**(9), 1027–37. (Chs. 4, 22)

Sheldon, F. T., Kavi, K. M., *et al.* (1992). Reliability measurement: from theory to practice. *IEEE Software*, **9**(4), 13–20. (Ch. 24)

Shlaer, S. and Mellor, S. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press. (Ch. 14)

Shneiderman, B. (1998). *Designing the User Interface, 3rd edn.* Reading, MA: Addison-Wesley. (Ch. 16)

Siegal, J. (1998). OMG overview: CORBA and the OMA in enterprise computing. *Comm. ACM*, **41**(10), 37–43. (Ch. 12)

Silberschatz, A., Galvin, P. B., *et al.* (2002). *Operating System Concepts, 6th edn.* New York: John Wiley & Sons. (Ch. 15)

Skonnard, A. and Gudgin, M. (2002). *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More.* Boston: Addison-Wesley. (Ch. 12)

Snyder, C. (2003). *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces.* San Francisco: Morgan Kaufmann. (Ch. 16)

Spafford, E. (1989). The Internet worm: crisis and aftermath. *Comm. ACM*, **32**(6), 678–87. (Ch. 3)

Spivey, J. M. (1990). Specifying a real-time kernel. *IEEE Software*, **7**(5), 21–8. (Ch. 10)

Spivey, J. M. (1992). *The Z Notation: A Reference Manual, 2nd edn.* London: Prentice Hall. (Chs. 10, 22)

Stal, M. (2002). Web services: beyond component-based computing. *Comm. ACM*, **45**(10), 71–6. (Ch. 12)

Stapleton, J. (1997). *DSDM Dynamic Systems Development Method.* Harlow: Addison-Wesley. (Ch. 17)

Stephens, M. and Rosenberg, D. (2003). *Extreme Programming Refactored.* Berkley, CA: Apress. (Ch. 17)

Stevens, P. and Pooley, R. (1999). *Software Engineering with Objects and Components.* Harlow: Addison-Wesley. (Ch. 6)

Storey, N. (1996). *Safety-Critical Computer Systems.* Harlow: Addison-Wesley. (Chs. 9, 20)

Suchman, L. (1983). Office procedures as practical action. *ACM Trans. on Office Information Systems*, **1**(3), 320–28. (Ch. 16)

Swartz, A. J. (1996). Airport 95: automated baggage system? *ACM Software Engineering Notes*, **21**(2), 79–83. (Ch. 2)

Symons, C. R. (1988). Function-point analysis: difficulties and improvements. *IEEE Trans. on Software Engineering*, **14**(1), 2–11. (Ch. 26)

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming, 2nd edn.* Harlow: Addison-Wesley. (Chs. 12, 19)

Tanenbaum, A. S. (2001). *Modern Operating Systems, 2nd edn.* Englewood Cliffs, NJ: Prentice Hall. (Ch. 15)

Thayer, R. H. (1997). Software system engineering: an engineering process. In *Software Requirements Engineering* (R. H. Thayer and M. Dorfmann, eds.). Los Alamitos: IEEE Computer Society Press, 84106. (Ch. 2)

Thayer, R. H. (2002). Software system engineering: a tutorial. *IEEE Computer*, **35**(4), 6873. (Ch. 2)

Tichy, W. (1985). RCS—a system for version control. *Software Practice and Experience*, **15**(7), 637–54. (Ch. 29)

Tracz, W. (2001). COTS myths and other lessons learned in component-based software development. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 99–112. (Ch. 18)

Turner, M., Budgen, D., *et al.* (2003). Turning software into a service. *IEEE Computer*, **36**(10), 38–45. (Ch. 12)

Ulrich, W. M. (1990). The evolutionary growth of software reengineering and the decade ahead. *American Programmer*, **3**(10), 14–20. (Ch. 21)

Vesperman, J. (2003). *Essential CVS*. Sebastopol, CA: O'Reilly & Associates. (Ch. 29)

Wall, L., Christiansen, T., *et al.* (1996). *Programming Perl*. Sebastopol, CA: O'Reilly & Associates. (Ch. 17)

Wang, N., Schmidt, D. C., *et al.* (2001). Overview of the CORBA component model. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 557–72. (Ch. 19)

Ward, P. and Mellor, S. (1985). *Structured Development for Real-Time Systems*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 8)

Warmer, J. and Kleppe, A. (1998). *The Object Constraint Language: Precise Modeling with UML*. Boston: Addison-Wesley. (Ch. 19)

Warren, I. (1998). *The Renaissance of Legacy Systems*. London: Springer. (Ch. 21)

Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand. (Chs. 17, 25)

Weinreich, R. and Sametinger, J. (2001). Component models and component services: concepts and principles. In *Component-Based Software Engineering* (G. T. Heineman and W. T. Councill, eds.). Boston: Addison-Wesley, 33–48. (Ch. 19)

Weiss, S. (2002). *Handheld Usability*. New York: John Wiley & Sons. (Ch. 16)

White, B. A. (2000). *Software Configuration Management Strategies and Rational ClearCase*. Reading, MA: Addison-Wesley. (Ch. 29)

White, S., Alford, M., *et al.* (1993). Systems engineering of computer-based systems. *IEEE Computer*, **26**(11), 54–65. (Ch. 2)

Whitgift, D. (1991). *Software Configuration Management: Methods and Tools*. Chichester: John Wiley & Sons. (Ch. 29)

Whittaker, J. W. (2002). *How to Break Software: A Practical Guide to Testing*. Boston: Addison-Wesley. (Ch. 23)

Williams, L., Kessler, R. R., *et al.* (2000). Strengthening the case for pair programming. *IEEE Software*, **17**(4), 19–25. (Ch. 17)

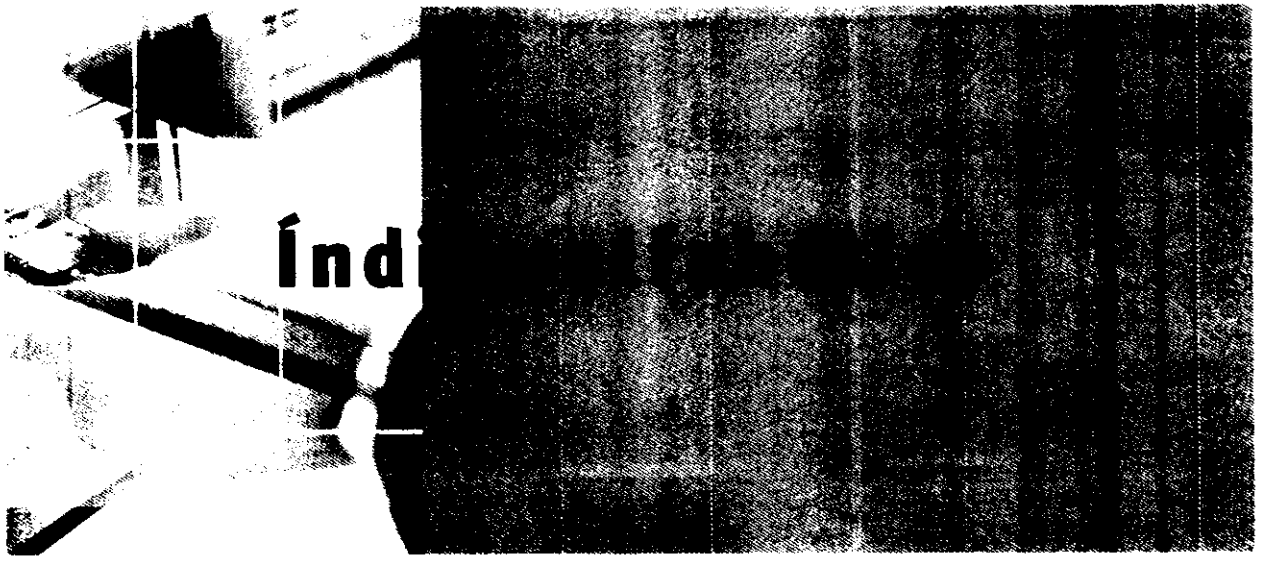
Wirfs-Brock, R. J. and Johnson, R. E. (1990). Surveying current research in object-oriented design. *Comm. ACM*, **33**(9), 104–24. (Ch. 18)

Wirfs-Brock, R., Wilkerson, B., *et al.* (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall. (Ch. 14)

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley. (Chs. 4, 9, 10, 22)

Wordsworth, J. B. (1991). The CICS application programming interface definition. *Proc. Z User Workshop*, Oxford, Berlin: Springer-Verlag. (Ch. 10)

Zimmermann, H. (1980). OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, **COM-28**(4), 425–32. (Ch. 11)



Índ

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X