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

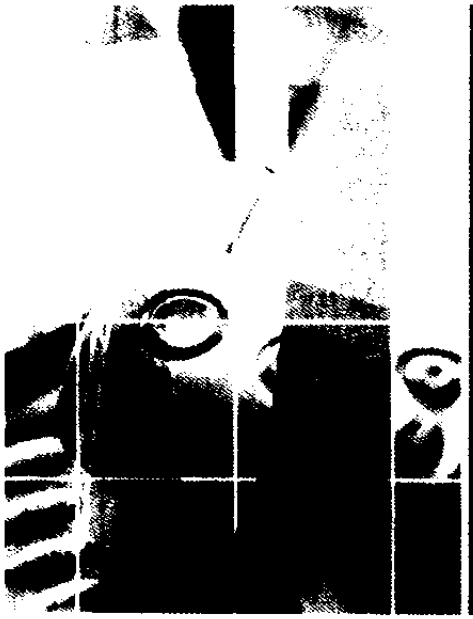
Diseño de cubierta: Equipo de diseño de Pearson Educación, S.A.

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



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

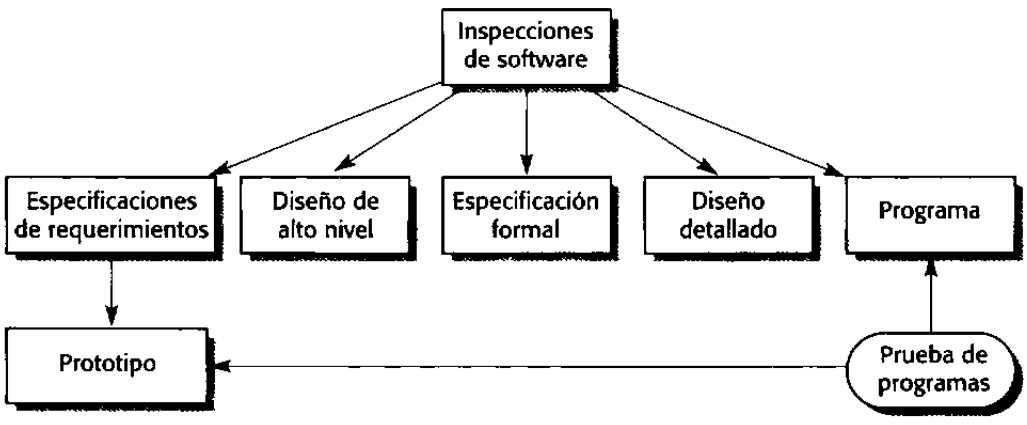


Figura 22.1 Verificación y validación estática y dinámica.

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.

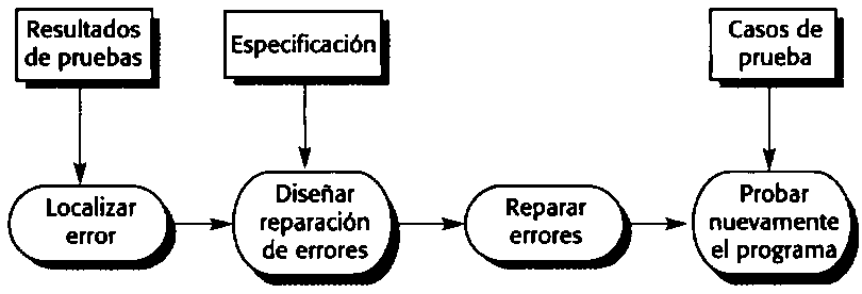


Figura 22.2 El proceso de depuración.

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

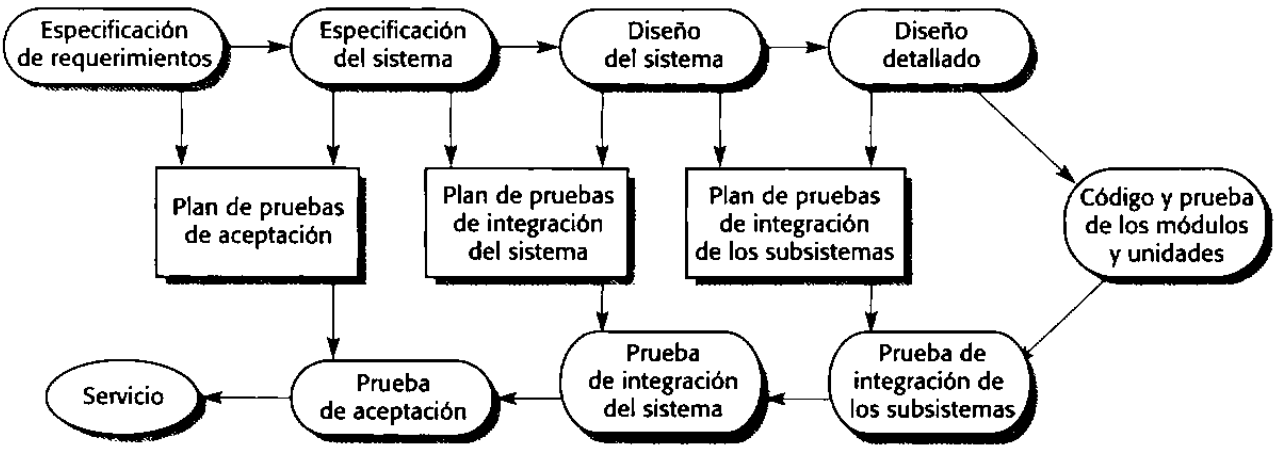


Figura 22.3 Planes de pruebas como un enlace entre las pruebas y el desarrollo.

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.

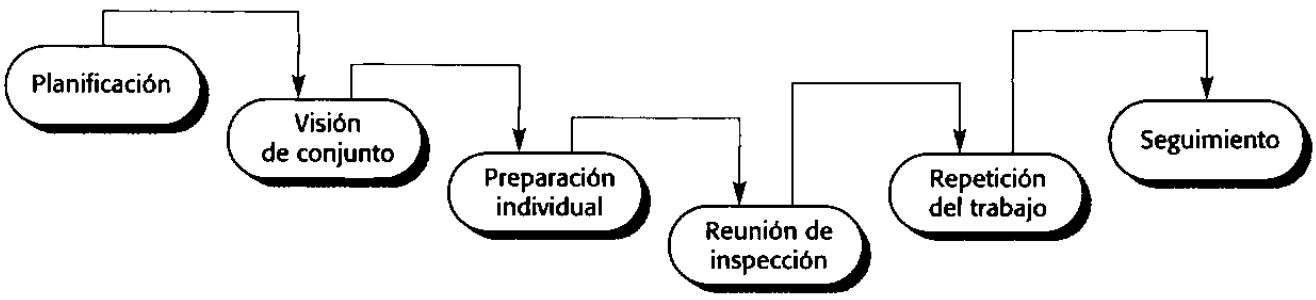


Figura 22.6 El proceso de inspección.

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.

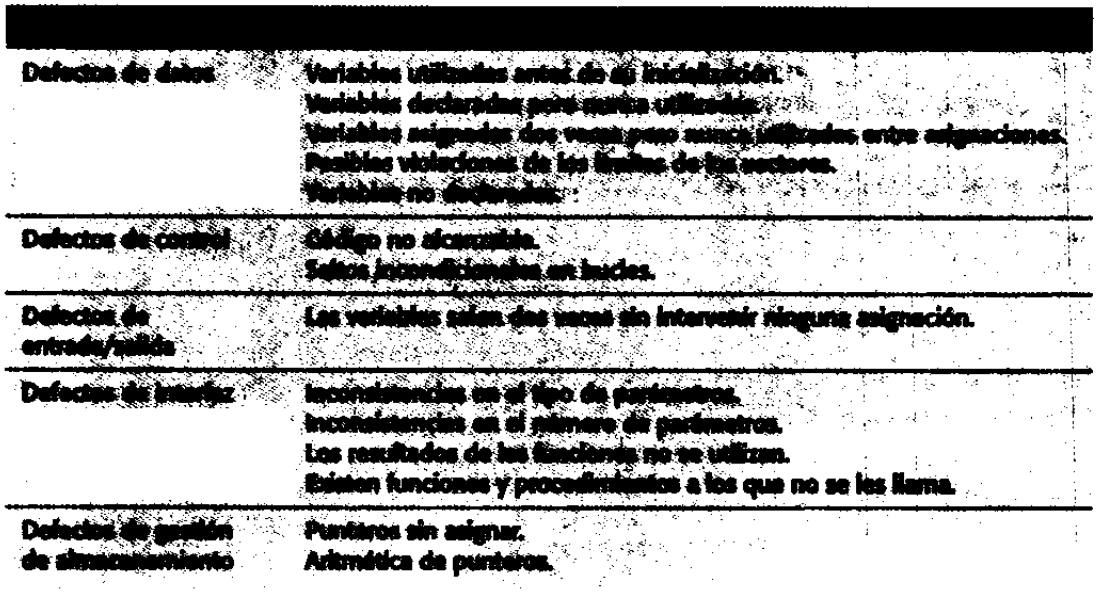


Figura 22.8 Comprobaciones del análisis estático automatizado.

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.

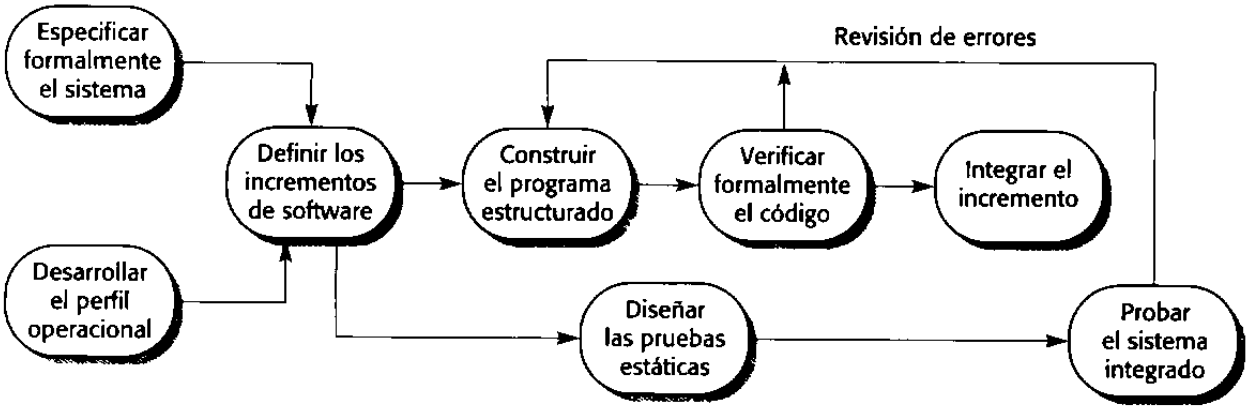


Figura 22.10 El proceso de desarrollo de Sala Limpia.

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.

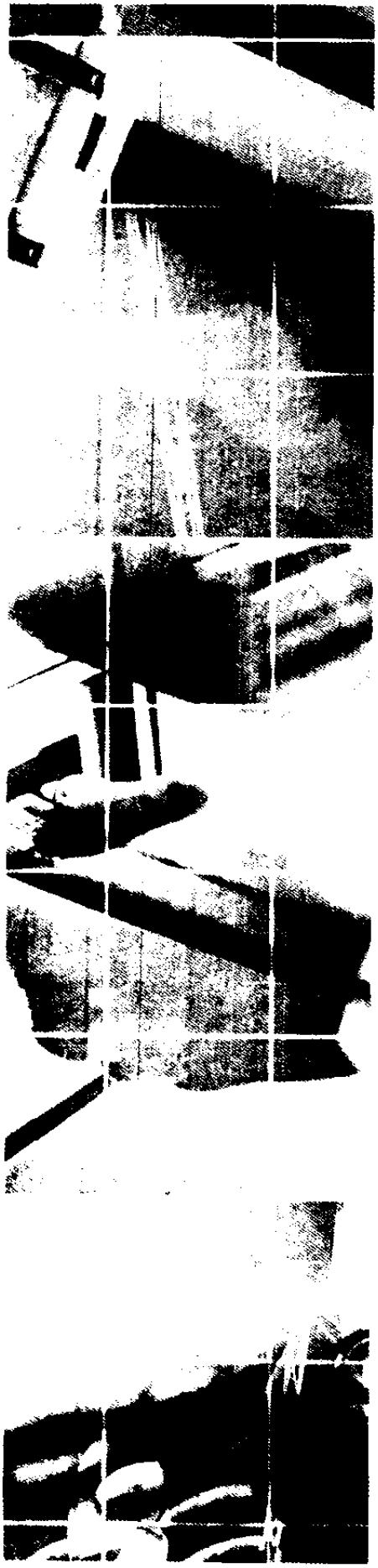
22.6 Utilizando su conocimiento de Java, C++, C o cualquier otro lenguaje de programación, derive una lista de comprobación de errores comunes (no errores sintácticos) que podrían no ser detectados por un compilador, pero que podrían ser detectados en una inspección de programas.

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-

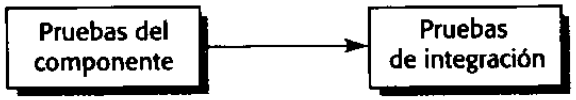


Figura 23.1
Fases de pruebas.

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-

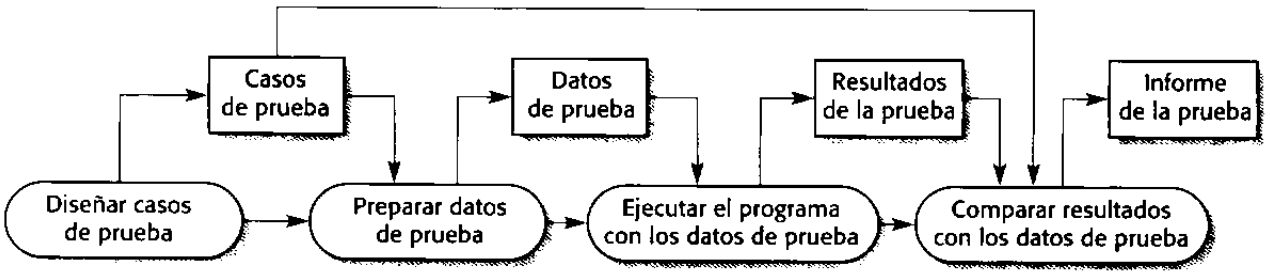


Figura 23.2 Un modelo del proceso de pruebas del software.

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

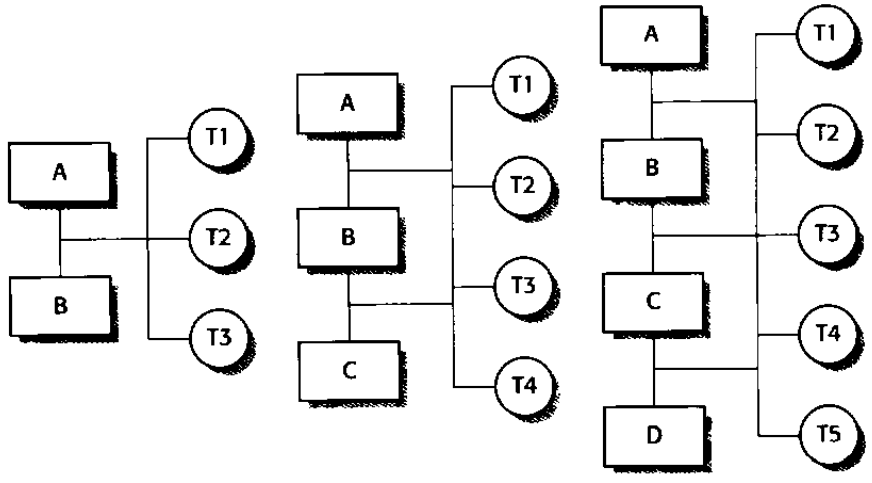


Figura 23.3 Pruebas de integración incrementales.

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

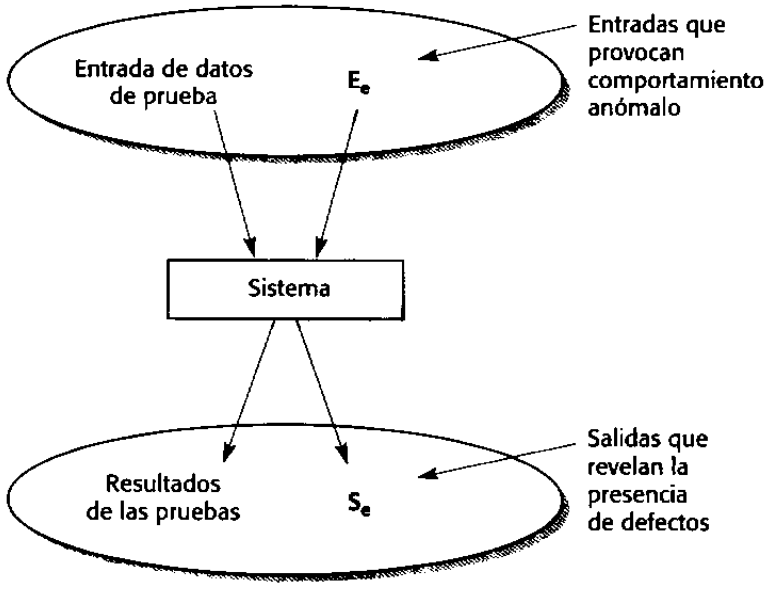


Figura 23.4 Pruebas de caja negra.

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**.

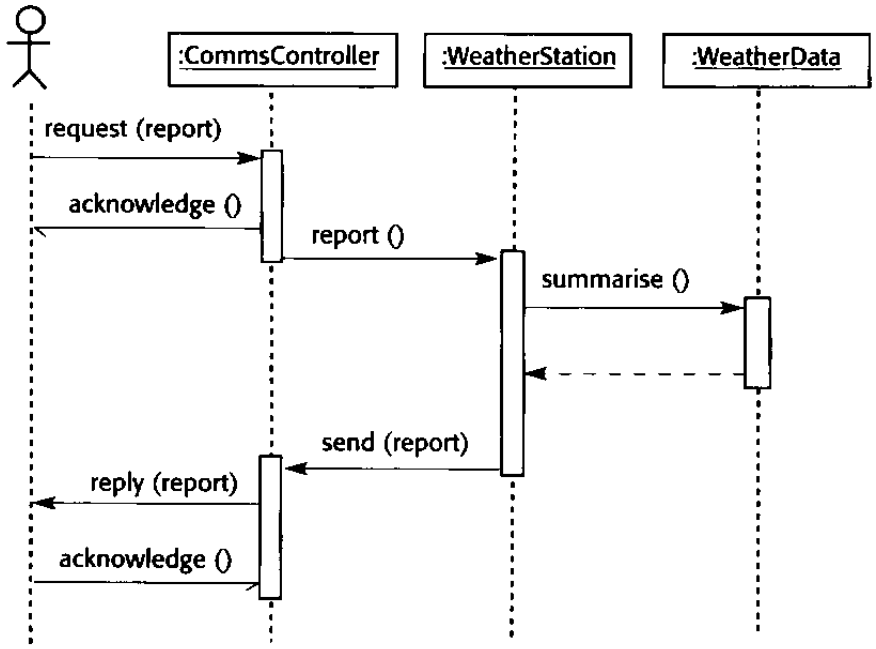


Figura 23.5 Diagrama de secuencia de recopilación de datos sobre el tiempo.

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-

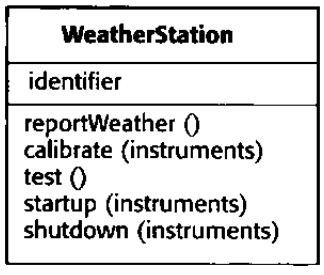


Figura 23.6
La interfaz del objeto de la estación meteorológica.

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

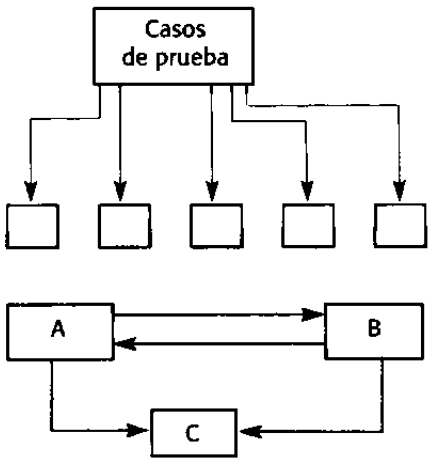


Figura 23.7 Pruebas de interfaces

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

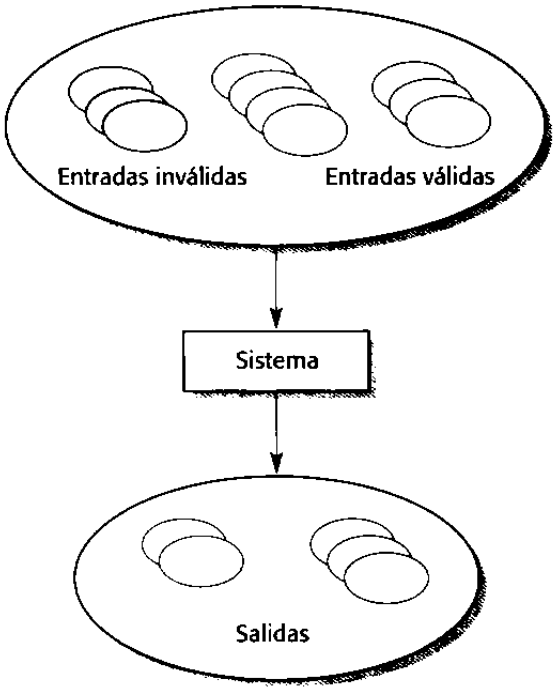


Figura 23.8
Particiones de equivalencia.

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.

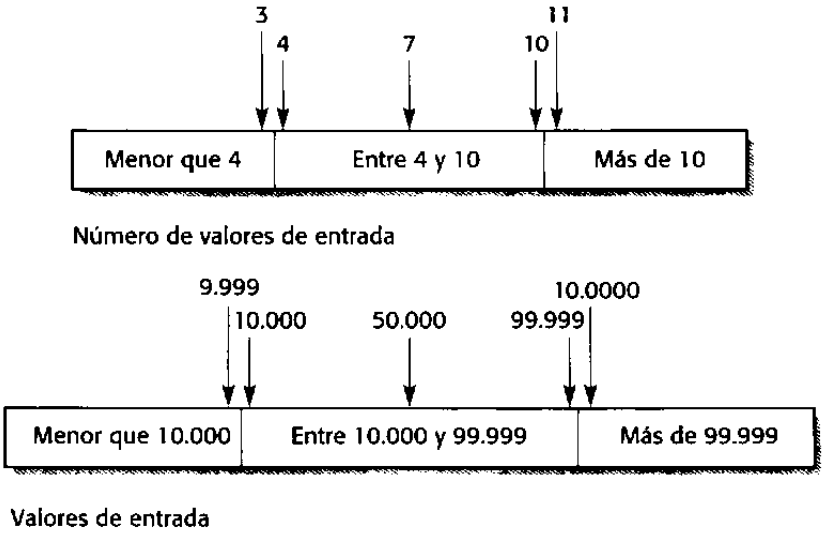


Figura 23.9
Ejemplos de particiones de equivalencia.

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-

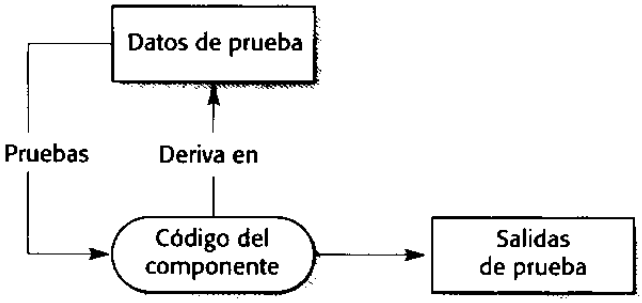


Figura 23.12 Pruebas estructurales.

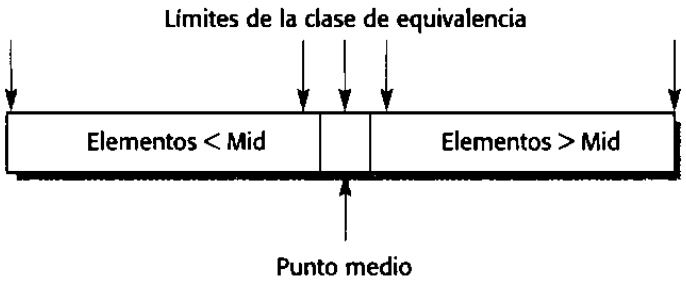


Figura 23.13 Clases de equivalencia de la búsqueda binaria.

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

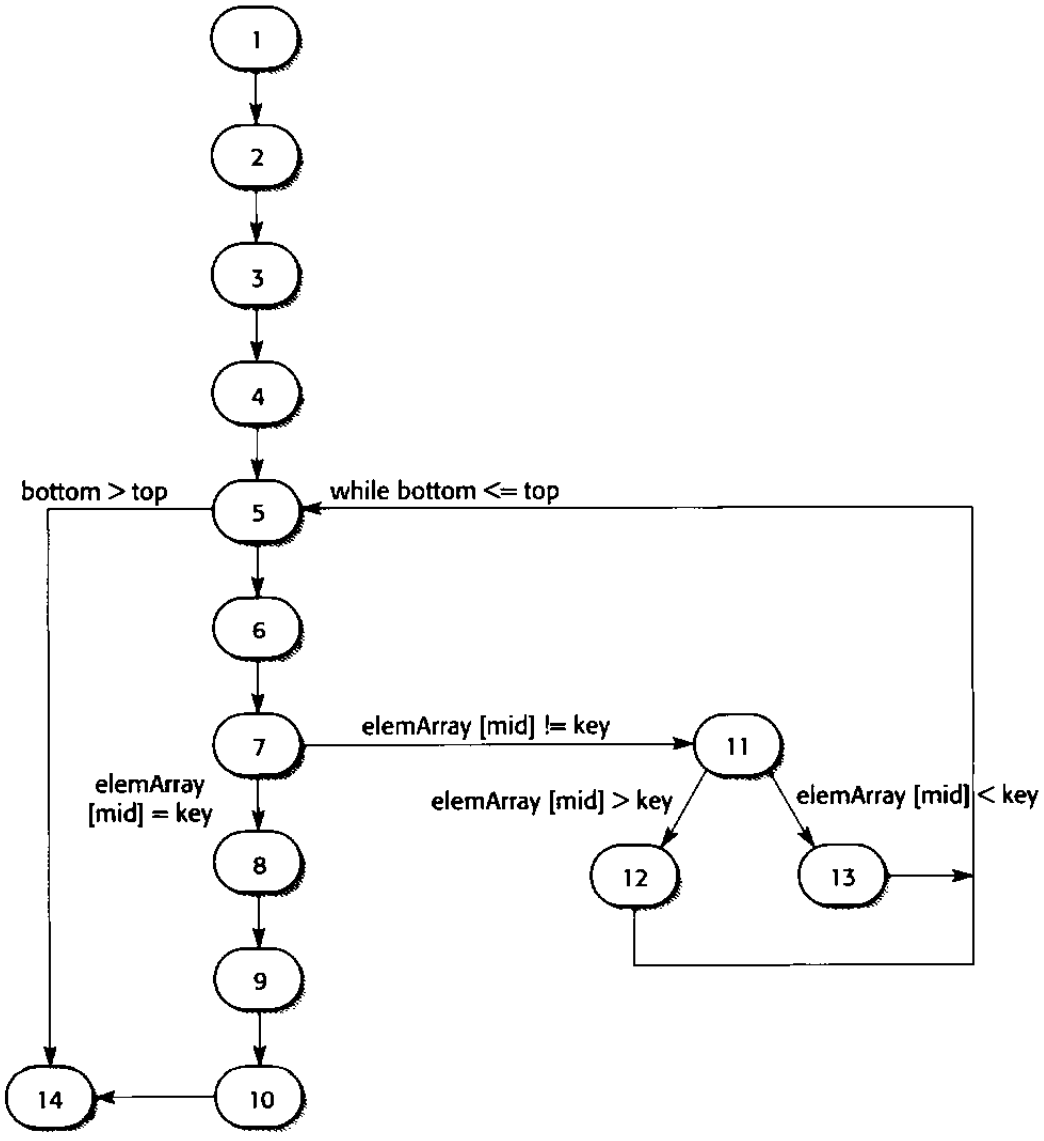


Figura 23.16
Grafo de flujo para una rutina de búsqueda binaria.

«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:

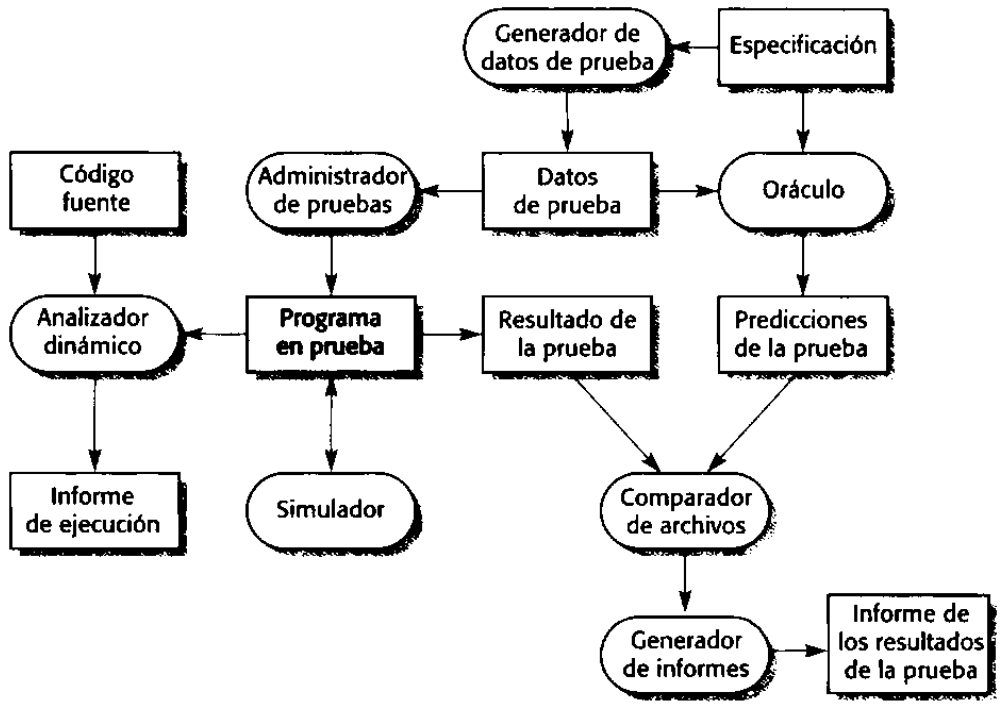


Figura 23.17
Un banco de trabajo de pruebas.

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.

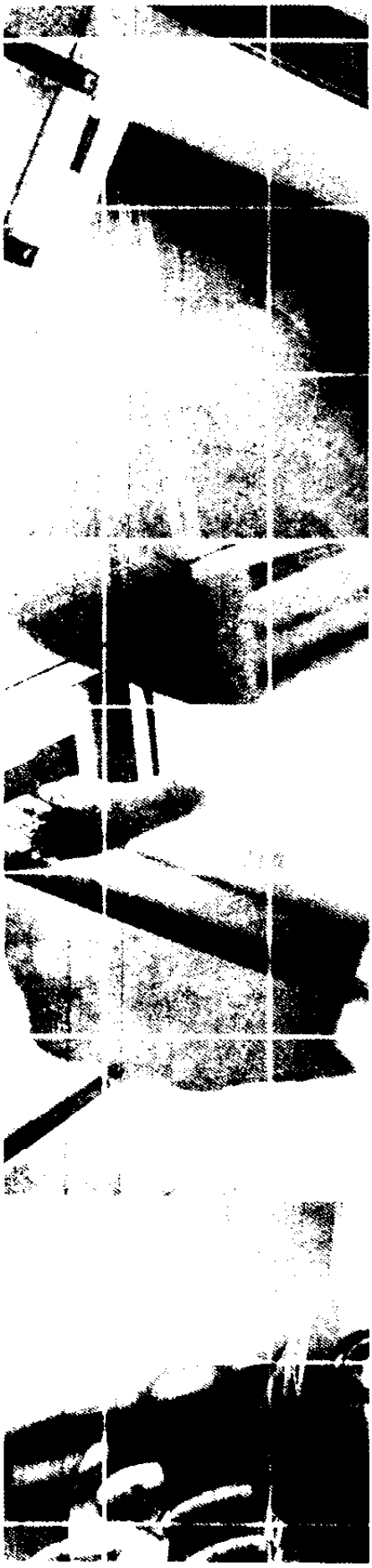
23.6 ¿Cuáles son los problemas que se plantean al desarrollar pruebas de rendimiento para un sistema de base de datos distribuida tal como el sistema LIBSYS?

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.

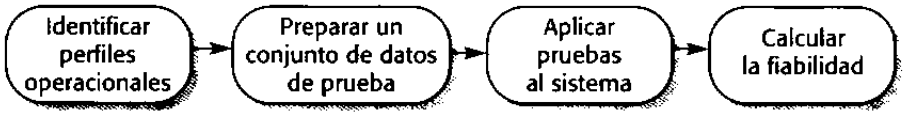


Figura 24.1 El proceso de medición de la fiabilidad.

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

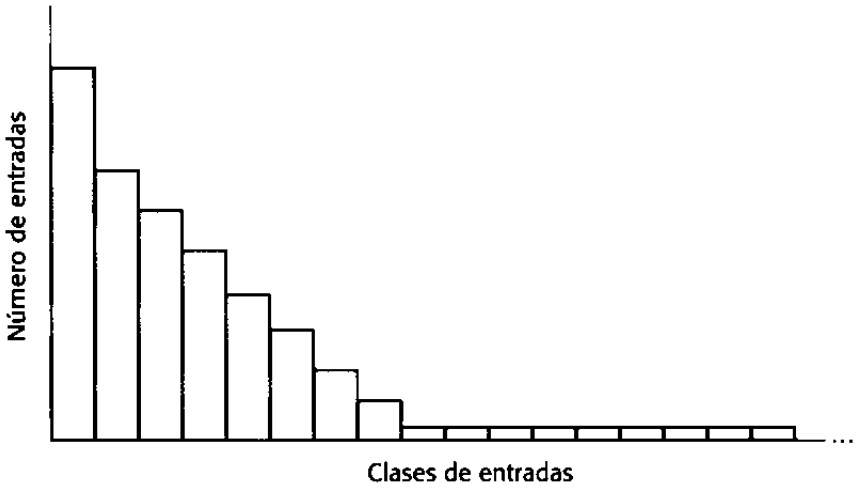


Figura 24.2
Un perfil operacional.

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-

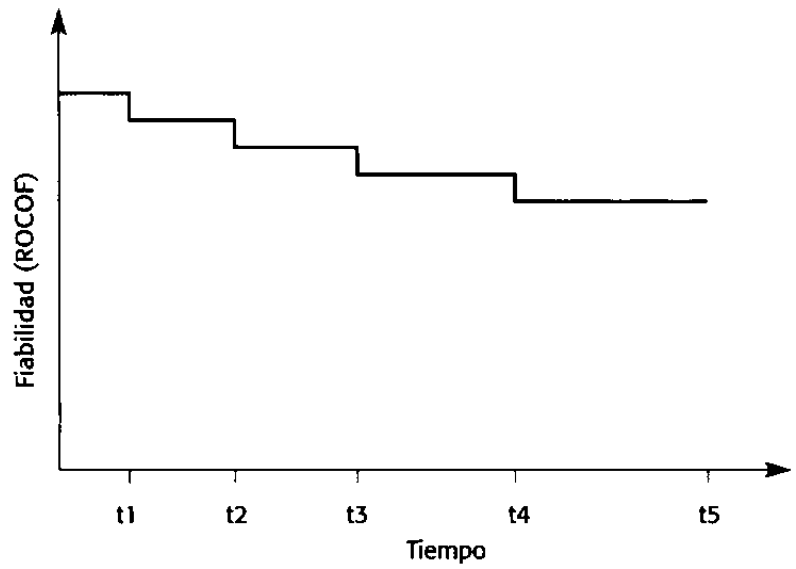


Figura 24.3 Modelo de función de pasos iguales del crecimiento de la fiabilidad.

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).

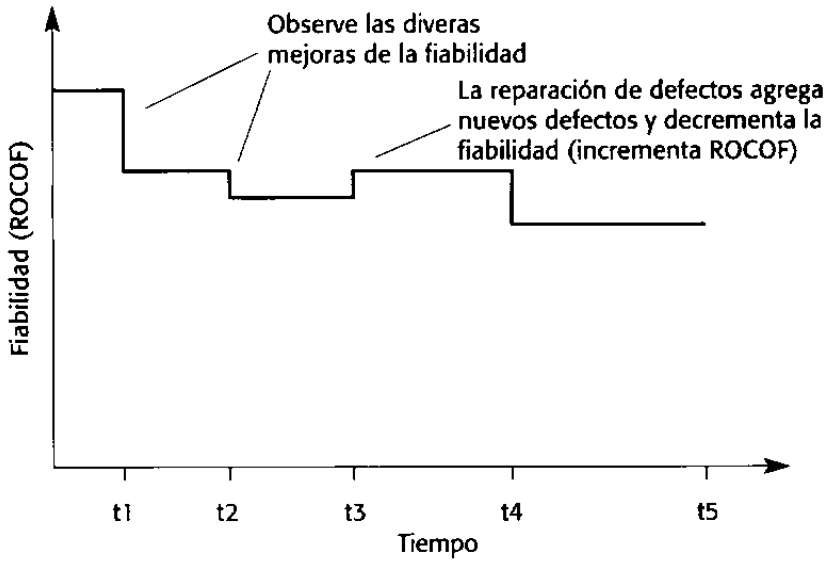


Figura 24.4
Modelo de función de pasos aleatorios del crecimiento de la fiabilidad.

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.

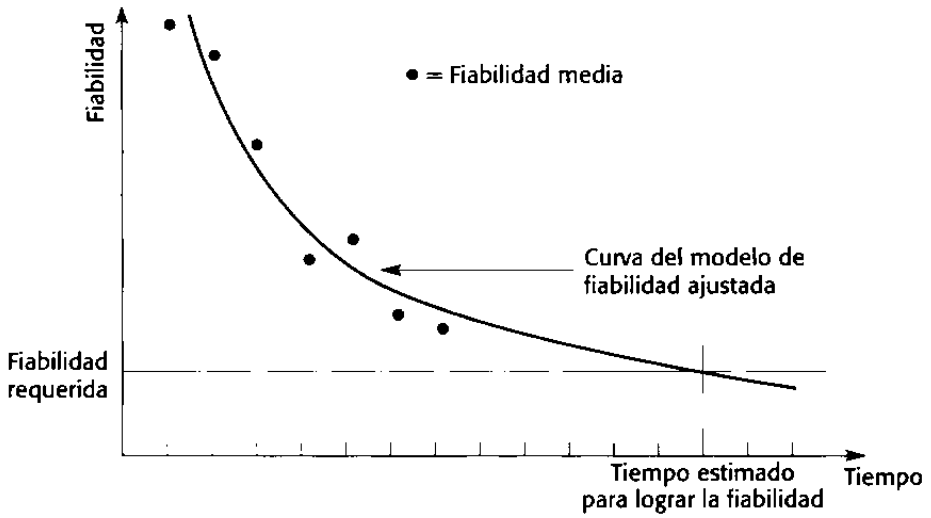


Figura 24.5
Predicción de la fiabilidad.

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

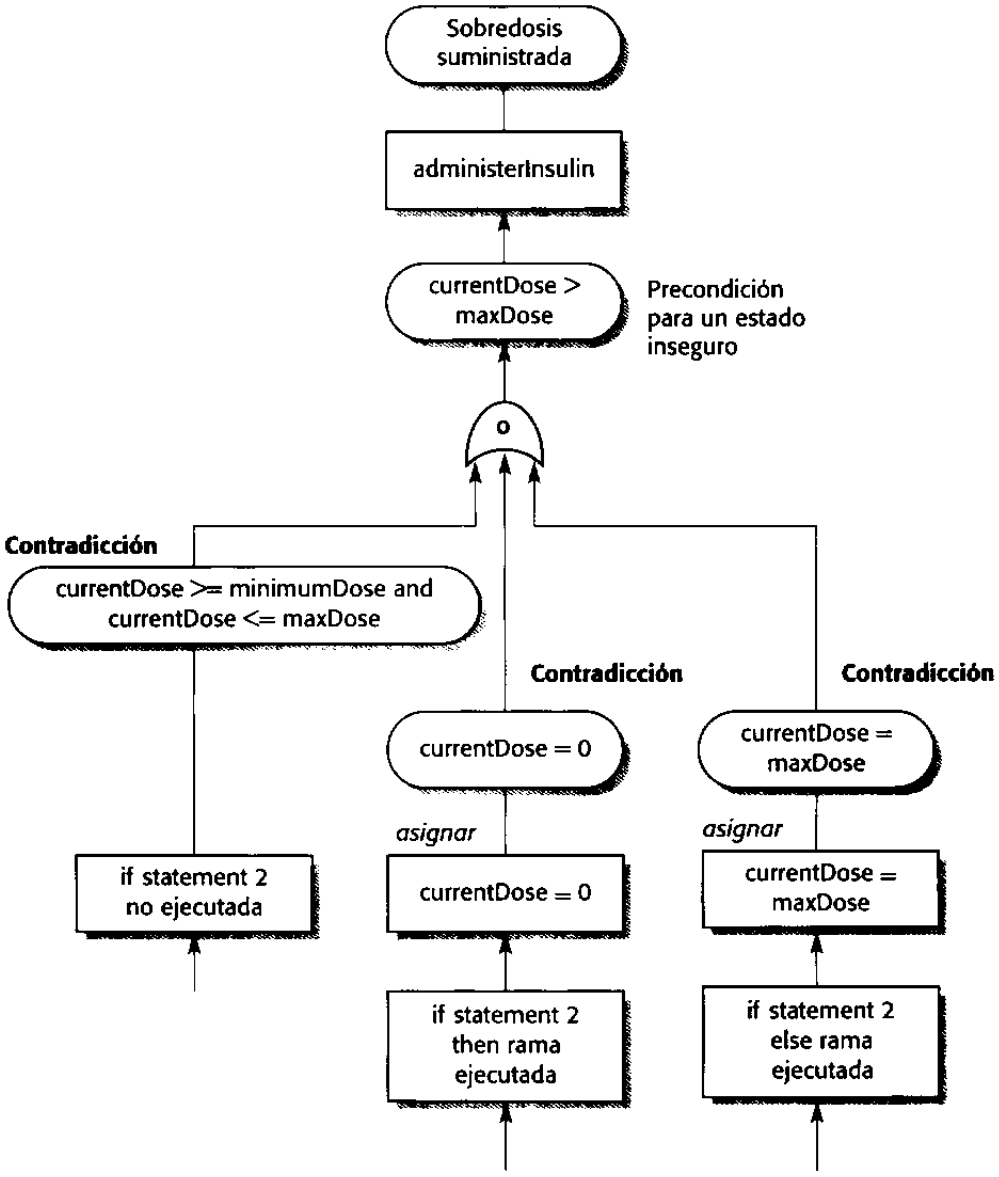


Figura 24.7
Argumento informal de seguridad basado en la demostración de contradicciones.

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: Imprese 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.

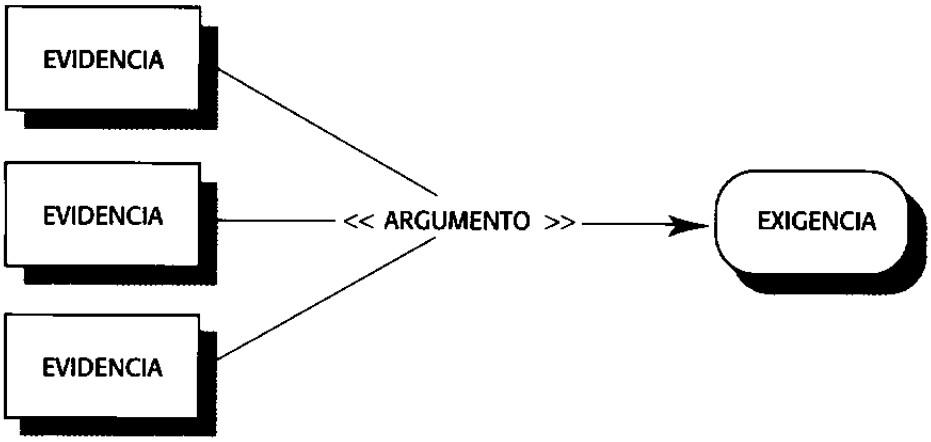


Figura 24.12 Estructura de un argumento.

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

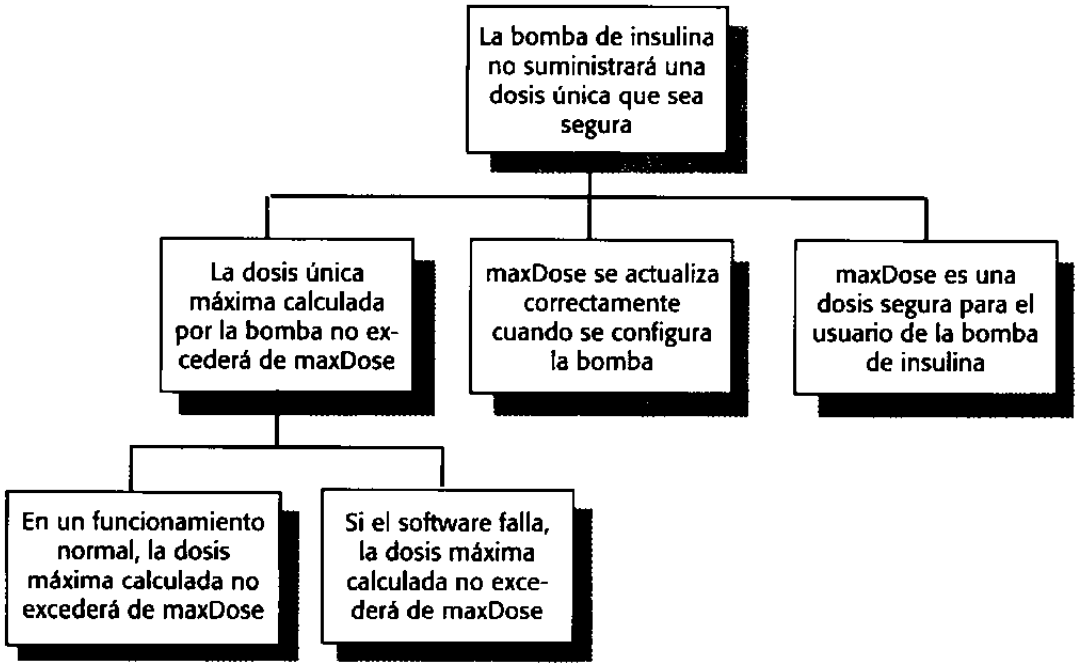


Figura 24.13 Jerarquía de exigencias en el argumento de seguridad para la bomba de insulina.

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.