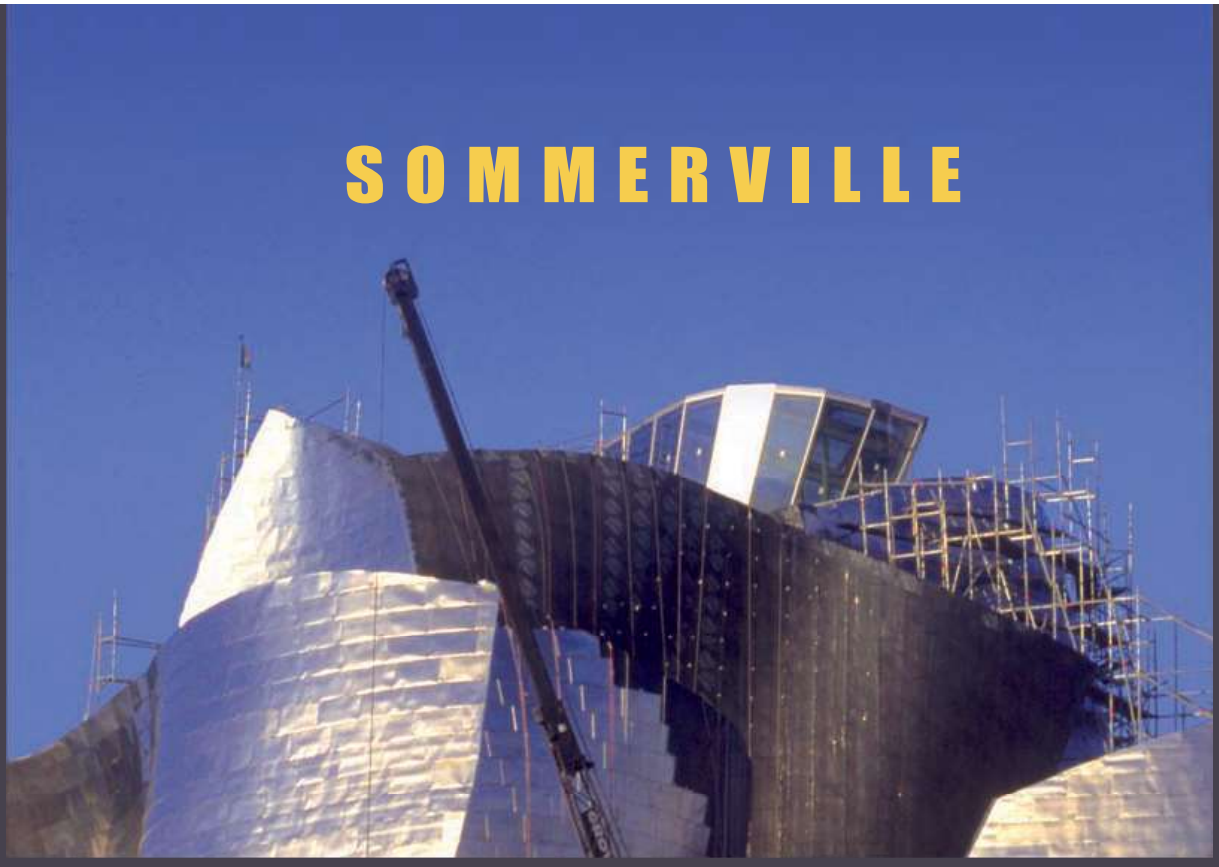
SOMMERVILLE



INGENIERÍA DE SOFTWARE

9

PEARSON

INGENIERÍA DE SOFTWARE

Novena edición

Ian Sommerville

Traducción:
Víctor Campos Olguín
Traductor especialista en Sistemas Computacionales

Revisión técnica:
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Miguel Ángel Torres Durán
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas-Instituto Politécnico Nacional, México

Darío Guillermo Cardacci
Universidad Abierta Interamericana, Buenos Aires, Argentina

Marcelo Martín Marciszack
Universidad Tecnológica Nacional, Córdoba, Argentina

Addison-Wesley

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Sommerville, Ian

Ingeniería de Software

PEARSON EDUCACIÓN, México, 2011

ISBN: 978-607-32-0603-7

Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 792

ISBN 9780137035151

Edición en español

Editor: Luis M. Cruz Castillo
e-mail: luis.cruz@pearson.com
Editor de desarrollo: Felipe Hernández Carrasco
Supervisor de producción: Juan José García Guzmán

NOVENA EDICIÓN, 2011

Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

ISBN VERSIÓN IMPRESA: 978-607-32-0603-7
ISBN VERSIÓN E-BOOK: 978-607-32-0604-4
ISBN E-CHAPTER: 978-607-32-0605-1

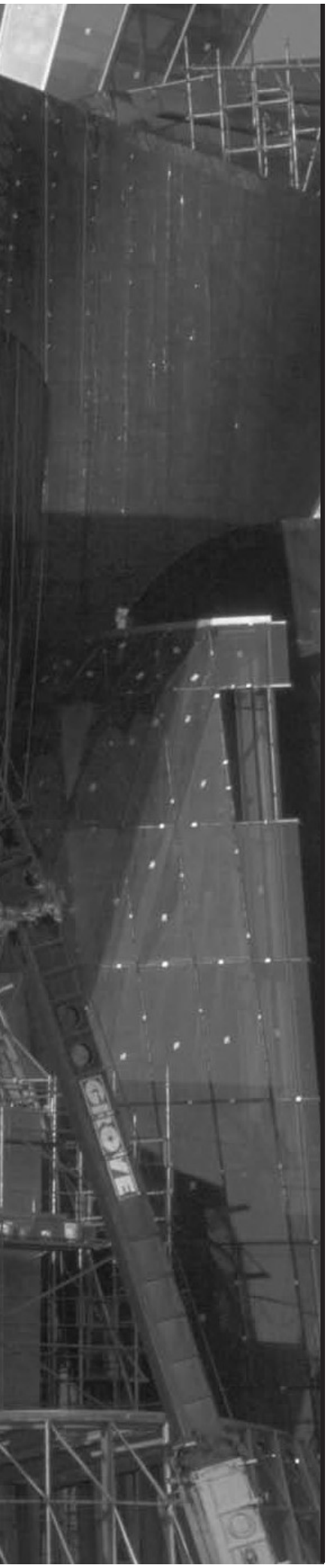
PRIMERA IMPRESIÓN

Impreso en México. *Printed in Mexico.*

1 2 3 4 5 6 7 8 9 0 - 14 13 12 11

Addison Wesley
es una marca de

PEARSON



8

Pruebas de software

Objetivos

El objetivo de este capítulo es introducirlo a las pruebas del software y los procesos necesarios para tales pruebas. Al estudiar este capítulo:

- comprenderá las etapas de las pruebas, desde las pruebas durante el desarrollo hasta la prueba de aceptación por los clientes del sistema;
- se introducirá en las técnicas que ayudan a elegir casos de prueba que se ponen en funcionamiento para descubrir los defectos del programa;
- entenderá el desarrollo de la primera prueba, donde se diseñan pruebas antes de escribir el código, las cuales operan automáticamente;
- conocerá las diferencias importantes entre pruebas de componente, de sistema y de liberación, y estará al tanto de los procesos y las técnicas de prueba del usuario.

Contenido

Las pruebas intentan demostrar que un programa hace lo que se intenta que haga, así como descubrir defectos en el programa antes de usarlo. Al probar el software, se ejecuta un programa con datos artificiales. Hay que verificar los resultados de la prueba que se opera para buscar errores, anomalías o información de atributos no funcionales del programa.

El proceso de prueba tiene dos metas distintas:

1. Demostrar al desarrollador y al cliente que el software cumple con los requerimientos. Para el software personalizado, esto significa que en el documento de requerimientos debe haber, por lo menos, una prueba por cada requerimiento. Para los productos de software genérico, esto quiere decir que tiene que haber pruebas para todas las características del sistema, junto con combinaciones de dichas características que se incorporarán en la liberación del producto.

2. Encontrar situaciones donde el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación. Tales situaciones son consecuencia de defectos del software. La prueba de defectos tiene la finalidad de erradicar el comportamiento indeseable del sistema, como caídas del sistema, interacciones indeseadas con otros sistemas, cálculos incorrectos y corrupción de datos.

La primera meta conduce a la prueba de validación; en ella, se espera que el sistema se desempeñe de manera correcta mediante un conjunto dado de casos de prueba, que refleje el uso previsto del sistema. La segunda meta se orienta a pruebas de defectos, donde los casos de prueba se diseñan para presentar los defectos. Los casos de prueba en las pruebas de defecto pueden ser deliberadamente confusos y no necesitan expresar cómo se usa normalmente el sistema. Desde luego, no hay frontera definida entre estos dos enfoques de pruebas. Durante las pruebas de validación, usted descubrirá defectos en el sistema; en tanto que durante las pruebas de defecto algunas de las pruebas demostrarán que el programa cumple con sus requerimientos.

El diagrama de la figura 8.1 ayuda a explicar las diferencias entre las pruebas de validación y de defecto. Piense en el sistema que va a probar como si fuera una caja negra. El sistema acepta entradas desde algún conjunto de entradas I y genera salidas en un conjunto de salidas O. Algunas de las salidas serán erróneas. Son las salidas en el conjunto O_e que el sistema genera en respuesta a las entradas en el conjunto I_e. La prioridad en las pruebas de defecto es encontrar dichas entradas en el conjunto I_e porque ellas revelan problemas con el sistema. Las pruebas de validación involucran pruebas con entradas correctas que están fuera de I_e y estimulan al sistema para generar las salidas correctas previstas.

Las pruebas no pueden demostrar que el software esté exento de defectos o que se comportará como se especifica en cualquier circunstancia. Siempre es posible que una prueba que usted pase por alto descubra más problemas con el sistema. Como afirma de forma elocuente Edsger Dijkstra, uno de los primeros contribuyentes al desarrollo de la ingeniería de software (Dijkstra *et al.*, 1972):

Las pruebas pueden mostrar sólo la presencia de errores, mas no su ausencia.

Las pruebas se consideran parte de un proceso más amplio de verificación y validación (V&V) del software. Aunque ambas no son lo mismo, se confunden con frecuencia. Barry

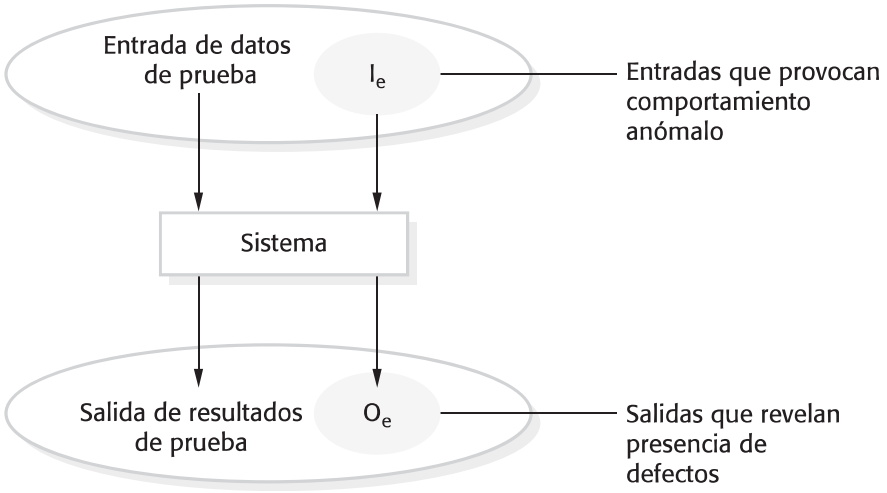


Figura 8.1 Modelo de entrada y salida de una prueba de programa

Boehm, pionero de la ingeniería de software, expresó de manera breve la diferencia entre las dos (Boehm, 1979):

- "Validación: ¿construimos el producto correcto?".
- "Verificación: ¿construimos bien el producto?".

Los procesos de verificación y validación buscan comprobar que el software por desarrollar cumpla con sus especificaciones, y brinde la funcionalidad deseada por las personas que pagan por el software. Dichos procesos de comprobación comienzan tan pronto como están disponibles los requerimientos y continúan a través de todas las etapas del proceso de desarrollo.

La finalidad de la verificación es comprobar que el software cumpla con su funcionalidad y con los requerimientos no funcionales establecidos. Sin embargo, la validación es un proceso más general. La meta de la validación es garantizar que el software cumpla con las expectativas del cliente. Va más allá del simple hecho de comprobar la conformidad con la especificación, para demostrar que el software hace lo que el cliente espera que haga. La validación es esencial pues, como se estudió en el capítulo 4, las especificaciones de requerimientos no siempre reflejan los deseos o las necesidades reales de los clientes y usuarios del sistema.

El objetivo final de los procesos de verificación y validación es establecer confianza de que el sistema de software es "adecuado". Esto significa que el sistema tiene que ser lo bastante eficaz para su uso esperado. El nivel de confianza adquirido depende tanto del propósito del sistema y las expectativas de los usuarios del sistema, como del entorno del mercado actual para el sistema:

1. *Propósito del software* Cuanto más crítico sea el software, más importante debe ser su confiabilidad. Por ejemplo, el nivel de confianza requerido para que se use el software en el control de un sistema crítico de seguridad es mucho mayor que el requerido para un prototipo desarrollado para demostrar nuevas ideas del producto.
2. *Expectativas del usuario* Debido a su experiencia con software no confiable y plagado de errores, muchos usuarios tienen pocas expectativas de la calidad del software, por lo que no se sorprenden cuando éste falla. Al instalar un sistema, los usuarios

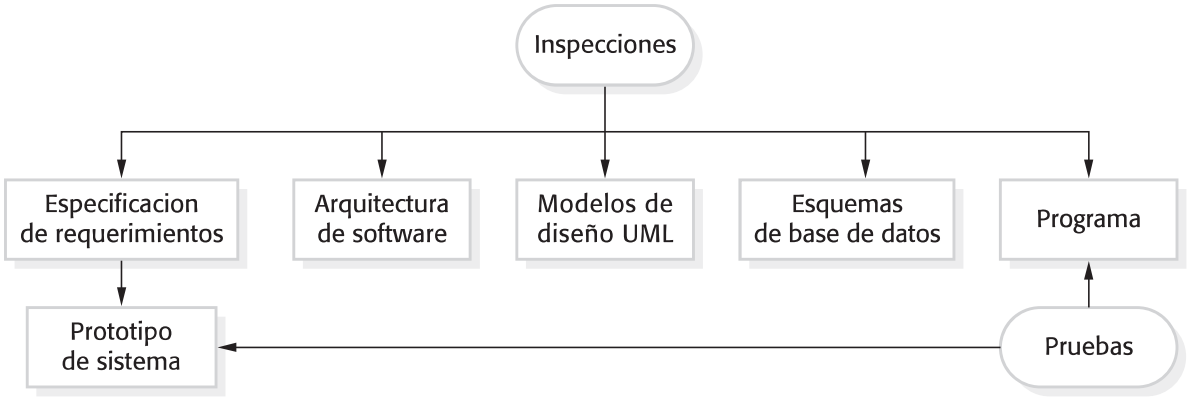


Figura 8.2 Inspecciones y pruebas

podrían soportar fallas, porque los beneficios del uso exceden los costos de la recuperación de fallas. Ante tales situaciones, no es necesario dedicar mucho tiempo en la puesta a prueba del software. Sin embargo, conforme el software se completa, los usuarios esperan que se torne más confiable, de modo que pueden requerirse pruebas exhaustivas en versiones posteriores.

3. *Entorno de mercado* Cuando un sistema se comercializa, los vendedores del sistema deben considerar los productos competitivos, el precio que los clientes están dispuestos a pagar por un sistema y la fecha requerida para entregar dicho sistema. En un ambiente competitivo, una compañía de software puede decidir lanzar al mercado un programa antes de estar plenamente probado y depurado, pues quiere que sus productos sean los primeros en ubicarse. Si un producto de software es muy económico, los usuarios tal vez toleren un nivel menor de fiabilidad.

Al igual que las pruebas de software, el proceso de verificación y validación implicaría inspecciones y revisiones de software. Estas últimas analizan y comprueban los requerimientos del sistema, los modelos de diseño, el código fuente del programa, e incluso las pruebas propuestas para el sistema. Éstas son las llamadas técnicas V&V "estáticas" donde no es necesario ejecutar el software para verificarlo. La figura 8.2 indica que las inspecciones y las pruebas del software soportan V&V en diferentes etapas del proceso del software. Las flechas señalan las etapas del proceso en que pueden usarse las técnicas.

Las inspecciones se enfocan principalmente en el código fuente de un sistema, aun cuando cualquier representación legible del software, como sus requerimientos o modelo de diseño, logre inspeccionarse. Cuando un sistema se inspecciona, se utiliza el conocimiento del sistema, su dominio de aplicación y el lenguaje de programación o modelado para descubrir errores.

Hay tres ventajas en la inspección del software sobre las pruebas:

1. Durante las pruebas, los errores pueden enmascarar (ocultar) otras fallas. Cuando un error lleva a salidas inesperadas, nunca se podrá asegurar si las anomalías de salida posteriores se deben a un nuevo error o son efectos colaterales del error original. Puesto que la inspección es un proceso estático, no hay que preocuparse por las interacciones entre errores. En consecuencia, una sola sesión de inspección descubriría muchos errores en un sistema.

Planeación de pruebas

La planeación de pruebas se interesa por la fecha y los recursos de todas las actividades durante el proceso de pruebas. Incluye la definición del proceso de pruebas, al tomar en cuenta tanto al personal como el tiempo disponible. Por lo general, se creará un plan de prueba que define lo que debe probarse, la fecha establecida de pruebas y cómo se registrarán éstas. Para sistemas críticos, el plan de prueba también puede incluir detalles de las pruebas que se van a correr en el software.

http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html

2. Las versiones incompletas de un sistema se pueden inspeccionar sin costos adicionales. Si un programa está incompleto, entonces es necesario desarrollar equipos de prueba especializados para poner a prueba las partes disponibles. Evidentemente, esto genera costos para el desarrollo del sistema.

3. Además de buscar defectos de programa, una inspección puede considerar también atributos más amplios de calidad de un programa, como el cumplimiento con estándares, la portabilidad y la mantenibilidad. Pueden buscarse ineficiencias, algoritmos inadecuados y estilos de programación imitados que hagan al sistema difícil de mantener y actualizar.

Las inspecciones de programa son una idea antigua y la mayoría de estudios y experimentos indican que las inspecciones son más efectivas para el descubrimiento de defectos, que para las pruebas del programa. Fagan (1986) reportó que más del 60% de los errores en un programa se detectan mediante inspecciones informales de programa. En el proceso de Cleanroom (cuarto limpio) (Prowell *et al*., 1999) se afirma que más del 90% de los defectos pueden detectarse en inspecciones del programa.

Sin embargo, las inspecciones no sustituyen las pruebas del software, ya que no son eficaces para descubrir defectos que surjan por interacciones inesperadas entre diferentes partes de un programa, problemas de temporización o dificultades con el rendimiento del sistema. Más aún, en compañías o grupos de desarrollo pequeños, suele ser especialmente difícil y costoso reunir a un equipo de inspección separado, ya que todos los miembros potenciales del equipo también podrían ser desarrolladores de software. En el capítulo 24 (“Gestión de la calidad”) se estudian las revisiones e inspecciones con más detenimiento. En el capítulo 15 se explica el análisis estático automatizado, en el cual el texto fuente de un programa se analiza automáticamente para descubrir anomalías. Este capítulo se enfoca en las pruebas y los procesos de pruebas.

La figura 8.3 presenta un modelo abstracto del proceso de prueba “tradicional”, como se utiliza en el desarrollo dirigido por un plan. Los casos de prueba son especificaciones de las entradas a la prueba y la salida esperada del sistema (los resultados de la prueba), además de información sobre lo que se pone a prueba. Los datos de prueba son las entradas que se diseñaron para probar un sistema. En ocasiones pueden generarse automáticamente datos de prueba; no obstante, es imposible la generación automática de casos de prueba, pues debe estar implicada gente que entienda lo que se supone que tiene que hacer el sistema para especificar los resultados de prueba previstos. Sin embargo, es posible automatizar la ejecución de pruebas. Los resultados previstos se comparan automáticamente con los resultados establecidos, de manera que no haya necesidad de que un individuo busque errores y anomalías al correr las pruebas.

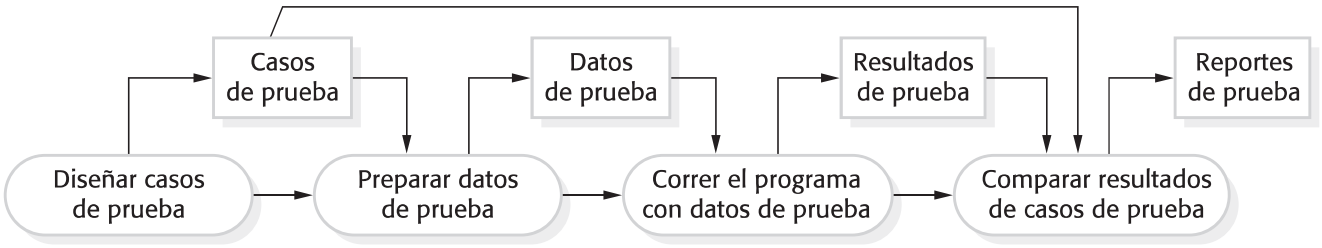


Figura 8.3 Modelo del proceso de pruebas de software

Por lo general, un sistema de software comercial debe pasar por tres etapas de pruebas:

1. Pruebas de desarrollo, donde el sistema se pone a prueba durante el proceso para descubrir errores (*bugs*) y defectos. Es probable que en el desarrollo de prueba intervengan diseñadores y programadores del sistema.
2. Versiones de prueba, donde un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios. La meta de la prueba de versión es comprobar que el sistema cumpla con los requerimientos de los participantes del sistema.
3. Pruebas de usuario, donde los usuarios reales o potenciales de un sistema prueban el sistema en su propio entorno. Para productos de software, el "usuario" puede ser un grupo interno de marketing, que decide si el software se comercializa, se lanza y se vende. Las pruebas de aceptación se efectúan cuando el cliente prueba de manera formal un sistema para decidir si debe aceptarse del proveedor del sistema, o si se requiere más desarrollo.

En la práctica, el proceso de prueba por lo general requiere una combinación de pruebas manuales y automatizadas. En las primeras pruebas manuales, un examinador opera el programa con algunos datos de prueba y compara los resultados con sus expectativas. Anota y reporta las discrepancias con los desarrolladores del programa. En las pruebas automatizadas, éstas se codifican en un programa que opera cada vez que se prueba el sistema en desarrollo. Comúnmente esto es más rápido que las pruebas manuales, sobre todo cuando incluye pruebas de regresión, es decir, aquellas que implican volver a correr pruebas anteriores para comprobar que los cambios al programa no introdujeron nuevos bugs.

El uso de pruebas automatizadas aumentó de manera considerable durante los últimos años. Sin embargo, las pruebas nunca pueden ser automatizadas por completo, ya que esta clase de pruebas sólo comprueban que un programa haga lo que supone que tiene que hacer. Es prácticamente imposible usar pruebas automatizadas para probar sistemas que dependan de cómo se ven las cosas (por ejemplo, una interfaz gráfica de usuario) o probar que un programa no presenta efectos colaterales indeseados.

8.1 Pruebas de desarrollo

Las pruebas de desarrollo incluyen todas las actividades de prueba que realiza el equipo que elabora el sistema. El examinador del software suele ser el programador que diseñó dicho software, aunque éste no es siempre el caso. Algunos procesos de desarrollo usan parejas de programador/examinador (Cusamano y Selby, 1998) donde cada programador

Depuración

Depuración (*debugging*) es el proceso para corregir los errores y problemas descubiertos por las pruebas. Al usar información de las pruebas del programa, los depuradores, para localizar y reparar el error del programa, emplean tanto su conocimiento del lenguaje de programación como el resultado que se espera de la prueba. Este proceso recibe con frecuencia apoyo de herramientas de depuración interactivas que brindan información adicional sobre la ejecución del programa.

http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html

tiene un examinador asociado que desarrolla pruebas y auxilia con el proceso de pruebas. Para sistemas críticos, puede usarse un proceso más formal, con un grupo de prueba independiente dentro del equipo de desarrollo. Son responsables del desarrollo de pruebas y del mantenimiento de registros detallados de los resultados de las pruebas.

Durante el desarrollo, las pruebas se realizan en tres niveles de granulación:

1. Pruebas de unidad, donde se ponen a prueba unidades de programa o clases de objetos individuales. Las pruebas de unidad deben enfocarse en comprobar la funcionalidad de objetos o métodos.
2. Pruebas del componente, donde muchas unidades individuales se integran para crear componentes compuestos. La prueba de componentes debe enfocarse en probar interfaces del componente.
3. Pruebas del sistema, donde algunos o todos los componentes en un sistema se integran y el sistema se prueba como un todo. Las pruebas del sistema deben enfocarse en poner a prueba las interacciones de los componentes.

Las pruebas de desarrollo son, ante todo, un proceso de prueba de defecto, en las cuales la meta consiste en descubrir bugs en el software. Por lo tanto, a menudo están entrelazadas con la depuración: el proceso de localizar problemas con el código y cambiar el programa para corregirlos.

8.1.1 Pruebas de unidad

Las pruebas de unidad son el proceso de probar componentes del programa, como métodos o clases de objetos. Las funciones o los métodos individuales son el tipo más simple de componente. Las pruebas deben llamarse para dichas rutinas con diferentes parámetros de entrada. Usted puede usar los enfoques para el diseño de casos de prueba que se estudian en la sección 8.1.2, con la finalidad de elaborar las pruebas de función o de método.

Cuando pone a prueba las clases de objetos, tiene que diseñar las pruebas para brindar cobertura a todas las características del objeto. Esto significa que debe:

- probar todas las operaciones asociadas con el objeto;
- establecer y verificar el valor de todos los atributos relacionados con el objeto;
- poner el objeto en todos los estados posibles. Esto quiere decir que tiene que simular todos los eventos que causen un cambio de estado.

Figura 8.4 Interfaz de objeto de estación meteorológica

Considere, por ejemplo, el objeto de estación meteorológica del modelo analizado en el capítulo 7. La interfaz de este objeto se muestra en la figura 8.4. Tiene un solo atributo, que es su identificador (*identifier*). Ésta es una constante que se establece cuando se instala la estación meteorológica. Por consiguiente, sólo se necesita una prueba que demuestre si se configuró de manera adecuada. Usted necesita definir casos de prueba para todos los métodos asociados con el objeto, como reportWeather, reportStatus, etcétera. Aunque lo ideal es poner a prueba los métodos en aislamiento, en algunos casos son precisas ciertas secuencias de prueba. Por ejemplo, para someter a prueba el método que desactiva los instrumentos de la estación meteorológica (*shutdown*), se necesita ejecutar el método *restart* (reinicio).

La generalización o herencia provoca que sea más complicada la prueba de las clases de objetos. Usted no debe poner únicamente a prueba una operación en la clase donde se definió, ni suponer que funcionará como se esperaba en las subclases que heredan la operación. La operación que se hereda puede hacer conjeturas sobre otras operaciones y atributos. Es posible que no sean válidas en algunas subclases que hereden la operación. Por consiguiente, tiene que poner a prueba la operación heredada en todos los contextos en que se utilice.

Para probar los estados de la estación meteorológica, se usa un modelo de estado, tal como el que se muestra en la figura 7.8 del capítulo anterior. Al usar este modelo, identificará secuencias de transiciones de estado que deban probarse y definirá secuencias de eventos para forzar dichas transiciones. En principio, hay que probar cualquier secuencia posible de transición de estado, aunque en la práctica ello resulte muy costoso. Los ejemplos de secuencias de estado que deben probarse en la estación meteorológica incluyen:

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Siempre que sea posible, se deben automatizar las pruebas de unidad. En estas pruebas de unidad automatizadas, podría usarse un marco de automatización de pruebas (como JUnit) para escribir y correr sus pruebas de programa. Los marcos de pruebas de unidad ofrecen clases de pruebas genéricas que se extienden para crear casos de prueba específicos. En tal caso, usted podrá correr todas las pruebas que implementó y reportar, con frecuencia mediante alguna GUI, el éxito o el fracaso de las pruebas. Es común que toda una serie de pruebas completa opere en algunos segundos, de modo que es posible ejecutar todas las pruebas cada vez que efectúe un cambio al programa.

Un conjunto automatizado de pruebas tiene tres partes:

1. Una parte de configuración, en la cual se inicializa el sistema con el caso de prueba, esto es, las entradas y salidas esperadas.

2. Una parte de llamada (*call*), en la cual se llama al objeto o al método que se va a probar.
3. Una parte de declaración, en la cual se compara el resultado de la llamada con el resultado esperado. Si la información se evalúa como verdadera, la prueba tuvo éxito; pero si resulta falsa, entonces fracasó.

En ocasiones, el objeto que se prueba tiene dependencias de otros objetos que tal vez no se escribieron o que, si se utilizan, frenan el proceso de pruebas. Si su objeto llama a una base de datos, por ejemplo, esto requeriría un proceso de configuración lento antes de usarse. En tales casos, usted puede decidir usar objetos *mock* (simulados). Éstos son objetos con la misma interfaz como los usados por objetos externos que simulan su funcionalidad. Por ende, un objeto mock que aparenta una base de datos suele tener sólo algunos ítems de datos que se organizan en un arreglo. Por lo tanto, puede entrar rápidamente a ellos, sin las sobrecargas de llamar a una base de datos y acceder a discos. De igual modo, los objetos mock pueden usarse para simular una operación anormal o eventos extraños. Por ejemplo, si se pretende que el sistema tome acción en ciertas horas del día, su objeto mock simplemente regresará estas horas, independientemente de la hora real en el reloj.

8.1.2 Elección de casos de pruebas de unidad

Las pruebas son costosas y consumen tiempo, así que es importante elegir casos efectivos de pruebas de unidad. La efectividad significa, en este caso, dos cuestiones:

1. Los casos de prueba tienen que mostrar que, cuando se usan como se esperaba, el componente que se somete a prueba hace lo que se supone que debe hacer.
2. Si hay defectos en el componente, éstos deberían revelarse mediante los casos de prueba.

En consecuencia, hay que escribir dos tipos de casos de prueba. El primero debe reflejar una operación normal de un programa y mostrar que el componente funciona. Por ejemplo, si usted va a probar un componente que crea e inicia el registro de un nuevo paciente, entonces, su caso de prueba debe mostrar que el registro existe en una base de datos, y que sus campos se configuraron como se especificó. El otro tipo de caso de prueba tiene que basarse en probar la experiencia de donde surgen problemas comunes. Debe usar entradas anormales para comprobar que se procesan de manera adecuada sin colapsar el componente.

Aquí se discuten dos estrategias posibles que serían efectivas para ayudarle a elegir casos de prueba. Se trata de:

1. Prueba de partición, donde se identifican grupos de entradas con características comunes y se procesan de la misma forma. Debe elegir las pruebas dentro de cada uno de dichos grupos.
2. Pruebas basadas en lineamientos, donde se usan lineamientos para elegir los casos de prueba. Dichos lineamientos reflejan la experiencia previa de los tipos de errores que suelen cometer los programadores al desarrollar componentes.

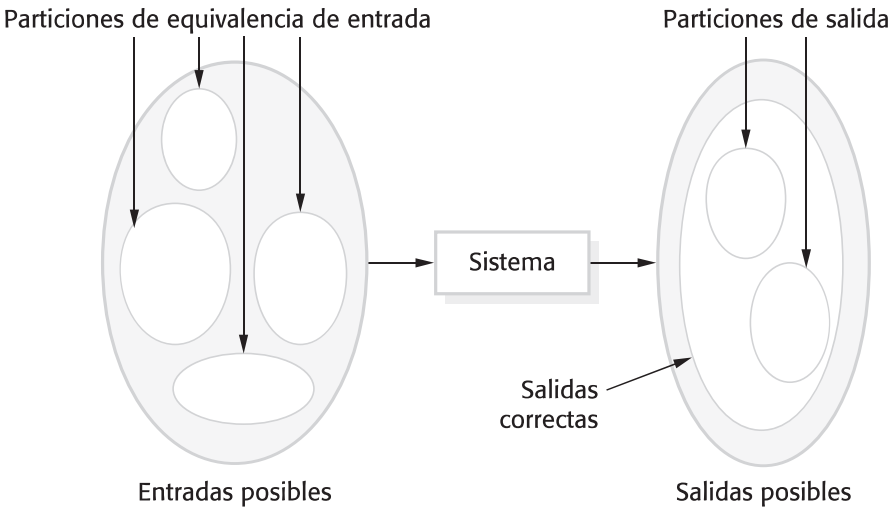


Figura 8.5 Partición de equivalencia

Los datos de entrada y los resultados de salida de un programa regularmente caen en un número de clases diferentes con características comunes. Los ejemplos de estas clases son números positivos, números negativos y selecciones de menú. Por lo general, los programas se comportan en una forma comparable a todos los miembros de una clase. Esto es, si usted prueba un programa que hace un cálculo y requiere dos números positivos, entonces esperaría que el programa se comportara de igual modo en todos los números positivos.

Debido a este comportamiento equivalente, dichas clases se llaman en ocasiones particiones de equivalencia o dominios (Bezier, 1990). Para el diseño de casos de prueba, un enfoque sistemático se basa en identificar todas las particiones de entrada y salida para un sistema o unos componentes. Los casos de prueba se elaboran de forma que las entradas o salidas se encuentren dentro de dichas particiones. La prueba de partición sirve para diseñar casos de prueba tanto para sistemas como para componentes.

En la figura 8.5, la elipse sombreada más grande ubicada en el lado izquierdo representa el conjunto de todas las entradas posibles al programa que se someterá a prueba. Las elipses más pequeñas sin sombrear constituyen particiones de equivalencia. Un programa que se ponga a prueba debe procesar de la misma forma todos los miembros de las particiones de equivalencia de entrada. Las particiones de equivalencia de salida son particiones dentro de las cuales todas las salidas tienen algo en común. En ocasiones hay un mapeo 1:1 entre particiones de equivalencia de entrada y salida. Sin embargo, éste no siempre es el caso; quizás usted necesite definir una partición de equivalencia de entrada independiente, donde la única característica común de las entradas sea que generan salidas dentro de la misma partición de salida. El área sombreada en la elipse izquierda representa excepciones que pueden ocurrir (es decir, respuestas a entradas inválidas).

Una vez identificado el conjunto de particiones, los casos de prueba se eligen de cada una de dichas particiones. Una buena regla empírica para la selección de casos de prueba es seleccionar casos de prueba en las fronteras de las particiones, además de casos cerca del punto medio de la partición. La razón es que diseñadores y programadores tienden a considerar valores típicos de entradas cuando se desarrolla un sistema. Éstos se prueban al elegir el punto medio de la partición. Los valores frontera son usualmente atípicos (por ejemplo, cero puede comportarse de manera diferente a otros números no negativos), de modo que a veces los desarrolladores los pasan por alto. Con frecuencia ocurren fallas del programa cuando se procesan estos valores atípicos.

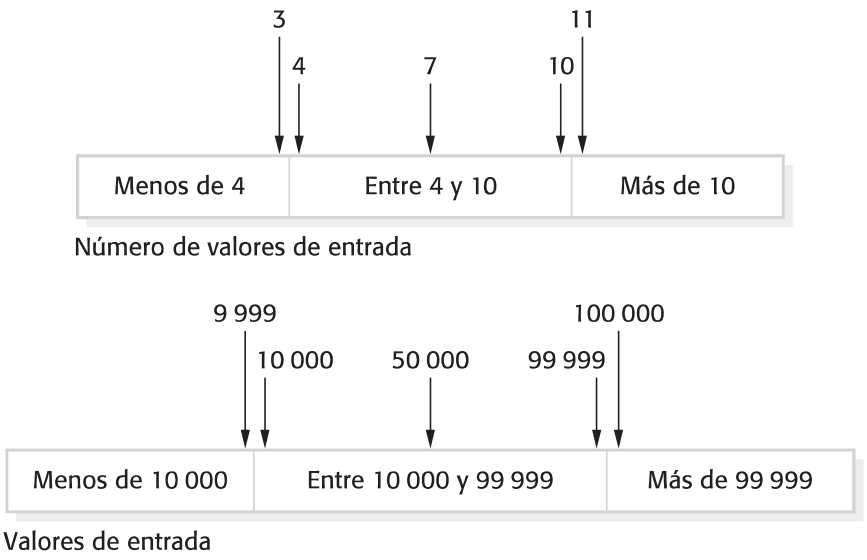


Figura 8.6 Particiones de equivalencia

Las particiones se identifican mediante la especificación del programa o la documentación del usuario y a partir de la experiencia, de donde se predicen las clases de valor de entrada que es probable que detecten errores. Por ejemplo, digamos que la especificación de un programa establece que el programa acepta de 4 a 8 entradas que son cinco dígitos enteros mayores que 10 000. Usted usa esta información para identificar las particiones de entrada y los posibles valores de entrada de prueba. Esto se muestra en la figura 8.6.

Cuando se usa la especificación de un sistema para reconocer particiones de equivalencia, se llama "pruebas de caja negra". Aquí no es necesario algún conocimiento de cómo funciona el sistema. Sin embargo, puede ser útil complementar las pruebas de caja negra con "pruebas de caja blanca", en las cuales se busca el código del programa para encontrar otras posibles pruebas. Por ejemplo, su código puede incluir excepciones para manejar las entradas incorrectas. Este conocimiento se utiliza para identificar "particiones de excepción": diferentes rangos donde deba aplicarse el mismo manejo de excepción.

La partición de equivalencia es un enfoque efectivo para las pruebas, porque ayuda a explicar los errores que cometen con frecuencia los programadores al procesar entradas en los bordes de las particiones. Usted también puede usar lineamientos de prueba para ayudarse a elegir casos de prueba. Los lineamientos encapsulan conocimiento sobre qué tipos de casos de prueba son efectivos para la detección de errores. Por ejemplo, cuando se prueban programas con secuencias, arreglos o listas, los lineamientos que pueden ayudar a revelar defectos incluyen:

1. Probar software con secuencias que tengan sólo un valor único. Los programadores naturalmente consideran a las secuencias como compuestas por muchos valores y, en ocasiones, incrustan esta suposición en sus programas. En consecuencia, si se presenta una secuencia de un valor único, es posible que un programa no funcione de manera adecuada.
2. Usar diferentes secuencias de diversos tamaños en distintas pruebas. Esto disminuye las oportunidades de que un programa con defectos genere accidentalmente una salida correcta, debido a algunas características accidentales de la entrada.
3. Derivar pruebas de modo que se acceda a los elementos primero, medio y último de la secuencia. Este enfoque revela problemas en las fronteras de la partición.

Pruebas de trayectoria

Las pruebas de trayectoria son una estrategia de prueba que se dirige principalmente a ejercitar cada trayectoria de ejecución independiente a través de un componente o programa. Si se ejecuta cualquier trayectoria independiente, entonces deben ejecutarse todos los enunciados en el componente al menos una vez. Todos los enunciados condicionales se prueban para los casos verdadero y falso. En un proceso de desarrollo orientado a objetos, la prueba de trayectoria puede usarse cuando se prueban los métodos asociados con los objetos.

http://www.SoftwareEngineering-9.com/Web/Testing/PathTest.html

El libro de Whittaker (2002) incluye muchos ejemplos de lineamientos que se pueden utilizar en el diseño de casos de prueba. Algunos de los lineamientos más generales que sugiere son:

- Elegir entradas que fuercen al sistema a generar todos los mensajes de error;
- Diseñar entradas que produzcan que los buffers de entrada se desborden;
- Repetir varias veces la misma entrada o serie de entradas;
- Forzar la generación de salidas inválidas;
- Forzar resultados de cálculo demasiado largos o demasiado pequeños.

Conforme adquiera experiencia con las pruebas, usted podrá desarrollar sus propios lineamientos sobre cómo elegir casos de prueba efectivos. En la siguiente sección de este capítulo se incluyen más ejemplos de lineamientos de prueba.

8.1.3 Pruebas de componentes

En general, los componentes de software son componentes compuestos constituidos por varios objetos en interacción. Por ejemplo, en el sistema de la estación meteorológica, el componente de reconfiguración incluye objetos que tratan con cada aspecto de la reconfiguración. El acceso a la funcionalidad de dichos objetos es a través de la interfaz de componente definida. Por consiguiente, la prueba de componentes compuestos tiene que enfocarse en mostrar que la interfaz de componente se comporta según su especificación. Usted puede suponer que dentro del componente se completaron las pruebas de unidad sobre el objeto individual.

La figura 8.7 ilustra la idea de la prueba de interfaz de componente. Suponga que los componentes A, B y C se integraron para crear un componente o subsistema más grande. Los casos de prueba no se aplican a los componentes individuales, sino más bien a la interfaz del componente compuesto, creado al combinar tales componentes. Los errores de interfaz en el componente compuesto quizá no se detecten al poner a prueba los objetos individuales, porque dichos errores resultan de interacciones entre los objetos en el componente.

Existen diferentes tipos de interfaz entre componentes de programa y, en consecuencia, distintos tipos de error de interfaz que llegan a ocurrir:

1. *Interfaces de parámetro* Son interfaces en que los datos, o en ocasiones referencias de función, pasan de un componente a otro. Los métodos en un objeto tienen una interfaz de parámetro.

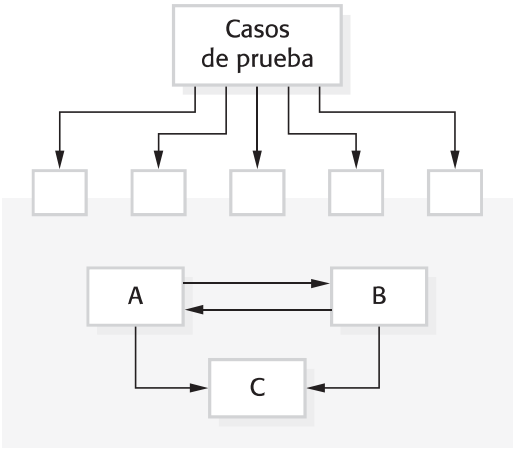


Figura 8.7 Prueba de interfaz

2. *Interfaces de memoria compartida* Son interfaces en que un bloque de memoria se reparte entre componentes. Los datos se colocan en la memoria de un subsistema y otros subsistemas los recuperan de ahí. Este tipo de interfaz se usa con frecuencia en sistemas embebidos, donde los sensores crean datos que se recuperan y son procesados por otros componentes del sistema.
3. *Interfaces de procedimiento* Son interfaces en que un componente encapsula un conjunto de procedimientos que pueden ser llamados por otros componentes. Los objetos y otros componentes reutilizables tienen esta forma de interfaz.
4. *Interfaces que pasan mensajes* Se trata de interfaces donde, al enviar un mensaje, un componente solicita un servicio de otro componente. El mensaje de retorno incluye los resultados para ejecutar el servicio. Algunos sistemas orientados a objetos tienen esta forma de interfaz, así como los sistemas cliente-servidor.

Los errores de interfaz son una de las formas más comunes de falla en los sistemas complejos (Lutz, 1993). Dichos errores caen en tres clases:

- *Uso incorrecto de interfaz* Un componente que llama a otro componente y comete algún error en el uso de su interfaz. Este tipo de error es común con interfaces de parámetro, donde los parámetros pueden ser del tipo equivocado, o bien, pasar en el orden o el número equivocados de parámetros.
- *Mala interpretación de interfaz* Un componente que malinterpreta la especificación de la interfaz del componente llamado y hace suposiciones sobre su comportamiento. El componente llamado no se comporta como se esperaba, lo cual entonces genera un comportamiento imprevisto en el componente que llama. Por ejemplo, un método de búsqueda binaria puede llamarse con un parámetro que es un arreglo desordenado. Entonces fallaría la búsqueda.
- *Errores de temporización* Ocurren en sistemas de tiempo real que usan una memoria compartida o una interfaz que pasa mensajes. El productor de datos y el consumidor de datos operan a diferentes niveles de rapidez. A menos que se tenga cuidado particular en el diseño de interfaz, el consumidor puede acceder a información obsoleta,

porque el productor de la información no actualizó la información de la interfaz compartida.

Las pruebas por defectos de interfaz son difíciles porque algunas fallas de interfaz sólo pueden manifestarse ante condiciones inusuales. Por ejemplo, se dice que un objeto implementa una cola como una estructura de datos de longitud fija. Un objeto que llama puede suponer que la cola se implementó como una estructura de datos infinita y no verificaría el desbordamiento de la cola, cuando se ingresa un ítem. Esta condición sólo se logra detectar durante las pruebas, al diseñar casos de prueba que fuercen el desbordamiento de la cola, y causen que el desbordamiento corrompa el comportamiento del objeto en cierta forma detectable.

Un problema posterior podría surgir derivado de interacciones entre fallas en diferentes módulos u objetos. Las fallas en un objeto sólo se detectan cuando algún otro objeto se comporta de una forma inesperada. Por ejemplo, un objeto llama a otro objeto para recibir algún servicio y supone que es correcta la respuesta; si el servicio de llamada es deficiente en algún modo, el valor devuelto puede ser válido pero equivocado. Esto no se detecta de inmediato, sino sólo se vuelve evidente cuando algún cálculo posterior sale mal.

Algunos lineamientos generales para las pruebas de interfaz son:

1. Examinar el código que se va a probar y listar explícitamente cada llamado a un componente externo. Diseñe un conjunto de pruebas donde los valores de los parámetros hacia los componentes externos estén en los extremos finales de sus rangos. Dichos valores extremos tienen más probabilidad de revelar inconsistencias de interfaz.
2. Donde los punteros pasen a través de una interfaz, pruebe siempre la interfaz con parámetros de puntero nulo.
3. Donde un componente se llame a través de una interfaz de procedimiento, diseñe pruebas que deliberadamente hagan que falle el componente. Diferir las suposiciones de falla es una de las interpretaciones de especificación equivocadas más comunes.
4. Use pruebas de esfuerzo en los sistemas que pasan mensajes. Esto significa que debe diseñar pruebas que generen muchos más mensajes de los que probablemente ocurran en la práctica. Ésta es una forma efectiva de revelar problemas de temporización.
5. Donde algunos componentes interactúen a través de memoria compartida, diseñe pruebas que varíen el orden en que se activan estos componentes. Tales pruebas pueden revelar suposiciones implícitas hechas por el programador, sobre el orden en que se producen y consumen los datos compartidos.

En ocasiones, las inspecciones y revisiones suelen ser más efectivas en costo que las pruebas para descubrir errores de interfaz. Las inspecciones pueden concentrarse en interfaces de componente e interrogantes sobre el comportamiento supuesto de la interfaz planteada durante el proceso de inspección. Un lenguaje robusto como Java permite que muchos errores de interfaz sean descubiertos por el compilador. Los analizadores estáticos (capítulo 15) son capaces de detectar un amplio rango de errores de interfaz.

Integración y pruebas incrementales

Las pruebas de sistema implican integrar diferentes componentes y, después, probar el sistema integrado que se creó. Siempre hay que usar un enfoque incremental para la integración y las pruebas (es decir, se debe incluir un componente, probar el sistema, integrar otro componente, probar de nuevo y así sucesivamente). Esto significa que, si ocurren problemas, quizá se deban a interacciones con el componente que se integró más recientemente.

La integración y las pruebas incrementales son fundamentales para los métodos ágiles como XP, donde las pruebas de regresión (véase sección 8.2) se efectúan cada vez que se integra un nuevo incremento.

http://www.SoftwareEngineering-9.com/Web/Testing/Integration.html

8.1.4 Pruebas del sistema

Las pruebas del sistema durante el desarrollo incluyen la integración de componentes para crear una versión del sistema y, luego, poner a prueba el sistema integrado. Las pruebas de sistema demuestran que los componentes son compatibles, que interactúan correctamente y que transfieren los datos correctos en el momento adecuado a través de sus interfaces. Evidentemente, se traslapan con las pruebas de componentes, pero existen dos importantes diferencias:

1. Durante las pruebas de sistema, los componentes reutilizables desarrollados por separado y los sistemas comerciales pueden integrarse con componentes desarrollados recientemente. Entonces se prueba el sistema completo.
2. Los componentes desarrollados por diferentes miembros del equipo o de grupos pueden integrarse en esta etapa. La prueba de sistema es un proceso colectivo más que individual. En algunas compañías, las pruebas del sistema implican un equipo de prueba independiente, sin la inclusión de diseñadores ni de programadores.

Cuando se integran componentes para crear un sistema, se obtiene un comportamiento emergente. Esto significa que algunos elementos de funcionalidad del sistema sólo se hacen evidentes cuando se reúnen los componentes. Éste podría ser un comportamiento emergente planeado que debe probarse. Por ejemplo, usted puede integrar un componente de autenticación con un componente que actualice información. De esta manera, tiene una característica de sistema que restringe la información actualizada de usuarios autorizados. Sin embargo, algunas veces, el comportamiento emergente no está planeado ni se desea. Hay que desarrollar pruebas que demuestren que el sistema sólo hace lo que se supone que debe hacer.

Por lo tanto, las pruebas del sistema deben enfocarse en poner a prueba las interacciones entre los componentes y los objetos que constituyen el sistema. También se prueban componentes o sistemas reutilizables para acreditar que al integrarse nuevos componentes funcionen como se esperaba. Esta prueba de interacción debe descubrir aquellos bugs de componente que sólo se revelan cuando lo usan otros componentes en el sistema. Las pruebas de interacción también ayudan a encontrar interpretaciones erróneas, cometidas por desarrolladores de componentes, acerca de otros componentes en el sistema.

Por su enfoque en las interacciones, las pruebas basadas en casos son un enfoque efectivo para la prueba del sistema. Normalmente, cada caso de uso es implementado por

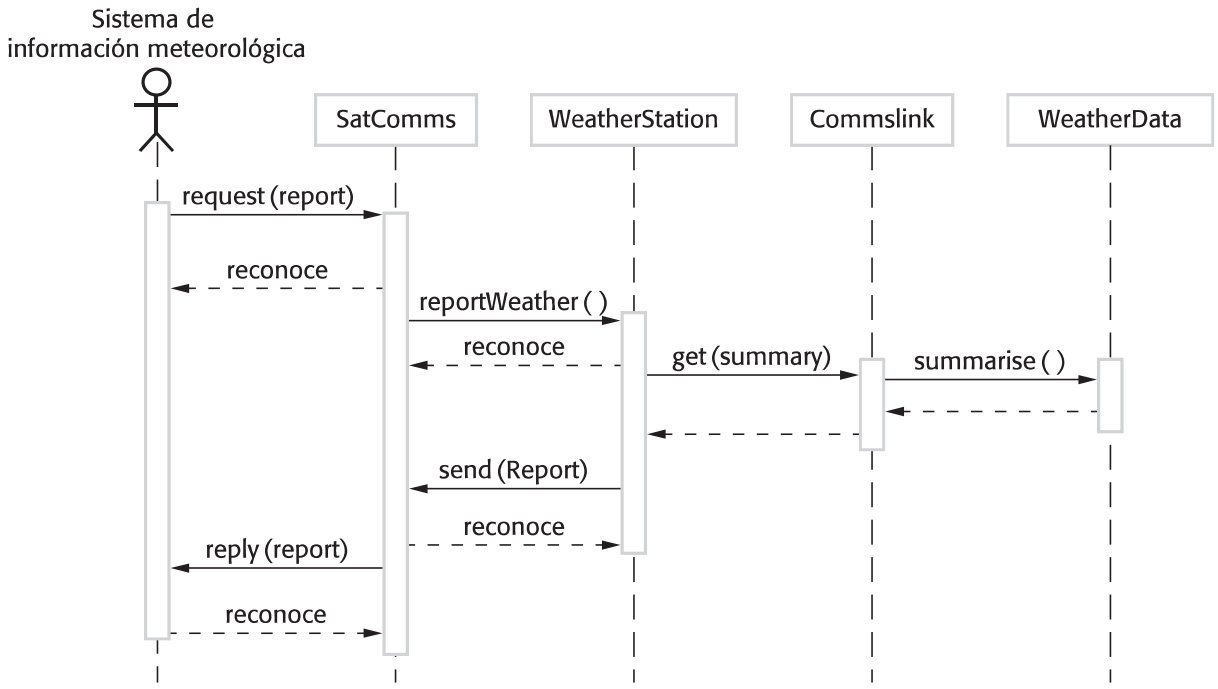


Figura 8.8 Gráfico de secuencia de recolección de datos meteorológicos

varios componentes u objetos en el sistema. Probar los casos de uso obliga a que ocurran estas interacciones. Si usted desarrolló un diagrama de secuencia para modelar la implementación de casos de uso, verá los objetos o componentes implicados en la interacción.

Para ilustrar lo anterior, se usa un ejemplo del sistema de estación meteorológica a campo abierto, donde se pide a la estación meteorológica reportar un resumen de datos a una computadora remota. El caso de uso para esto se describe en la figura 7.3 (capítulo anterior). La figura 8.8 (copia de la figura 7.7) muestra la secuencia de operaciones en la estación meteorológica, al responder a una petición de recolección de datos para el sistema de mapeo. Este diagrama sirve para identificar operaciones que se probarán y para ayudar a diseñar los casos de prueba para efectuar las pruebas. Por consiguiente, emitir una petición para un reporte dará como resultado la ejecución de la siguiente cadena de métodos:

SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize

El diagrama de secuencia ayuda a diseñar los casos de prueba específicos necesarios, pues muestra cuáles entradas se requieren y cuáles salidas se crean:

1. Una entrada de una petición para un reporte tiene que contar con reconocimiento asociado. En última instancia, a partir de la petición debe regresarse un reporte. Durante las pruebas, se debe crear un resumen de datos que sirva para comprobar que el reporte se organiza correctamente.
2. Una petición de entrada para un reporte a WeatherStation da como resultado la generación de un reporte resumido. Usted puede probar esto en aislamiento, creando datos brutos correspondientes al resumen que preparó para la prueba de SatComms, y demostrar que el objeto WeatherStation produce este resumen. Tales datos brutos se usan también para probar el objeto WeatherData.

Desde luego, en la figura 8.8 se simplificó el diagrama de secuencia para que no muestre excepciones. Asimismo, una prueba completa de caso/escenario de uso considera esto y garantiza que los objetos manejen adecuadamente las excepciones.

Para la mayoría de sistemas es difícil saber cuántas pruebas de sistemas son esenciales y cuándo hay que dejar de hacer pruebas. Las pruebas exhaustivas, donde se pone a prueba cada secuencia posible de ejecución del programa, son imposibles. Por lo tanto, las pruebas deben basarse en un subconjunto de probables casos de prueba. De manera ideal, para elegir este subconjunto, las compañías de software cuentan con políticas, las cuales pueden basarse en políticas de prueba generales, como una política de que todos los enunciados del programa se ejecuten al menos una vez. Como alternativa, pueden basarse en la experiencia de uso de sistema y, a la vez, enfocarse en probar las características del sistema operativo. Por ejemplo:

1. Tienen que probarse todas las funciones del sistema que se ingresen a través de un menú.
2. Debe experimentarse la combinación de funciones (por ejemplo, formateo de texto) que se ingrese por medio del mismo menú.
3. Donde se proporcione entrada del usuario, hay que probar todas las funciones, ya sea con entrada correcta o incorrecta.

Por experiencia con los principales productos de software, como procesadores de texto u hojas de cálculo, es claro que lineamientos similares se usan por lo general durante la prueba del producto. Usualmente funcionan cuando las características del software se usan en aislamiento. Los problemas se presentan, dice Whittaker (2002), cuando las combinaciones de características de uso menos común no se prueban en conjunto. Él da el ejemplo de cómo, en un procesador de texto de uso común, el uso de notas al pie de página con una plantilla en columnas múltiples causa la distribución incorrecta del texto.

Las pruebas automatizadas del sistema suelen ser más difíciles que las pruebas automatizadas de unidad o componente. Las pruebas automatizadas de unidad se apoyan en la predicción de salidas y, luego, en la codificación de dichas predicciones en un programa. En tal caso, se compara el pronóstico con el resultado. Sin embargo, el punto de aplicar un sistema puede ser generar salidas que sean grandes o no logren predecirse con facilidad. Se tiene que examinar una salida y demostrar su credibilidad sin crearla necesariamente por adelantado.

8.2 Desarrollo dirigido por pruebas

El desarrollo dirigido por pruebas (TDD, por las siglas de *Test-Driven Development*) es un enfoque al diseño de programas donde se entrelazan el desarrollo de pruebas y el de código (Beck, 2002; Jeffries y Melnik, 2007). En esencia, el código se desarrolla incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba. El desarrollo dirigido por pruebas se introdujo como parte de los métodos ágiles como la programación extrema. No obstante, se puede usar también en los procesos de desarrollo basados en un plan.

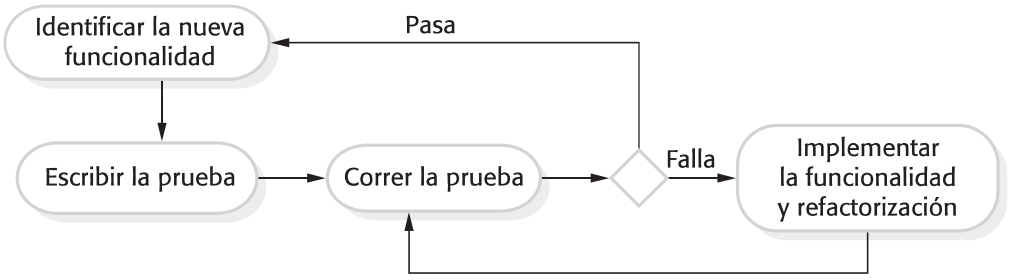


Figura 8.9 Desarrollo dirigido por pruebas

En la figura 8.9 se ilustra el proceso TDD fundamental. Los pasos en el proceso son los siguientes:

1. Se comienza por identificar el incremento de funcionalidad requerido. Éste usualmente debe ser pequeño y aplicable en pocas líneas del código.
2. Se escribe una prueba para esta funcionalidad y se implementa como una prueba automatizada. Esto significa que la prueba puede ejecutarse y reportarse, sin importar si aprueba o falla.
3. Luego se corre la prueba, junto con todas las otras pruebas que se implementaron. Inicialmente, no se aplica la funcionalidad, de modo que la nueva prueba fallará. Esto es deliberado, pues muestra que la prueba añade algo al conjunto de pruebas.
4. Luego se implementa la funcionalidad y se opera nuevamente la prueba. Esto puede incluir la refactorización del código existente, para perfeccionarlo y adicionar nuevo código a lo ya existente.
5. Una vez puestas en funcionamiento con éxito todas las pruebas, se avanza a la implementación de la siguiente funcionalidad.

Un entorno automatizado de pruebas, como el entorno JUnit que soporta pruebas del programa Java (Massol y Husted, 2003), es esencial para TDD. Conforme el código se desarrolla en incrementos muy pequeños, uno tiene la posibilidad de correr cada prueba, cada vez que se adiciona funcionalidad o se refactoriza el programa. Por consiguiente, las pruebas se incrustan en un programa independiente que corre las pruebas y apela al sistema que se prueba. Al usar este enfoque, en unos cuantos segundos se efectúan cientos de pruebas independientes.

Un argumento consistente con el desarrollo dirigido por pruebas es que ayuda a los programadores a aclarar sus ideas acerca de lo que realmente debe hacer un segmento de código. Para escribir una prueba, es preciso entender lo que se quiere, pues esta comprensión facilita la escritura del código requerido. Desde luego, si el conocimiento o la comprensión son incompletos, entonces no ayudará el desarrollo dirigido por pruebas. Por ejemplo, si su cálculo implica división, debería comprobar que no divide los números entre cero. En caso de que olvide escribir una prueba para esto, en el programa nunca se incluirá el código a comprobar.

Además de la mejor comprensión del problema, otros beneficios del desarrollo dirigido por pruebas son:

1. *Cobertura de código* En principio, cualquier segmento de código que escriba debe tener al menos una prueba asociada. Por lo tanto, puede estar seguro de que cual-

quier código en el sistema se ejecuta realmente. El código se prueba a medida que se escribe, de modo que los defectos se descubren con oportunidad en el proceso de desarrollo.

2. *Pruebas de regresión* Un conjunto de pruebas se desarrolla incrementalmente conforme se desarrolla un programa. Siempre es posible correr pruebas de regresión para demostrar que los cambios al programa no introdujeron nuevos bugs.

3. *Depuración simplificada* Cuando falla una prueba, debe ser evidente dónde yace el problema. Es preciso comprobar y modificar el código recién escrito. No se requieren herramientas de depuración para localizar el problema. Los reportes del uso del desarrollo dirigido por pruebas indican que difícilmente alguna vez se necesitará usar un depurador automatizado en el desarrollo dirigido por pruebas (Martin, 2007).

4. *Documentación del sistema* Las pruebas en sí actúan como una forma de documentación que describen lo que debe hacer el código. Leer las pruebas suele facilitar la comprensión del código.

Uno de los beneficios más importantes del desarrollo dirigido por pruebas es que reduce los costos de las pruebas de regresión. Estas últimas implican correr los conjuntos de pruebas ejecutadas exitosamente después de realizar cambios a un sistema. La prueba de regresión verifica que dichos cambios no hayan introducido nuevos bugs en el sistema, y que el nuevo código interactúa como se esperaba con el código existente. Las pruebas de regresión son muy costosas y, por lo general, poco prácticas cuando un sistema se prueba manualmente, pues son muy elevados los costos en tiempo y esfuerzo. Ante tales situaciones, usted debe ensayar y elegir las pruebas más relevantes para volver a correrlas, y es fácil perder pruebas importantes.

Sin embargo, las pruebas automatizadas, que son fundamentales para el desarrollo de primera prueba, reducen drásticamente los costos de las pruebas de regresión. Las pruebas existentes pueden volverse a correr de manera más rápida y menos costosa. Después de realizar cambios a un sistema en el desarrollo de la primera prueba, todas las pruebas existentes deben correr con éxito antes de añadir cualquier funcionalidad accesoria. Como programador, usted podría estar seguro de que la nueva funcionalidad que agregue no causará ni revelará problemas con el código existente.

El desarrollo dirigido por pruebas se usa más en el diseño de software nuevo, donde la funcionalidad se implementa en código nuevo o usa librerías estándar perfectamente probadas. Si se reutilizan grandes componentes en código o sistemas heredados, entonces se necesita escribir pruebas para dichos sistemas como un todo. El desarrollo dirigido por pruebas también puede ser ineficaz con sistemas multihilo. Los diferentes hilos pueden entrelazarse en diferentes momentos y en diversas corridas de pruebas y, por lo tanto, producirán resultados variados.

Si se usa el desarrollo dirigido por pruebas, se necesitará de un proceso de prueba del sistema para validar el sistema; esto es, comprobar que cumple con los requerimientos de todos los participantes del sistema. Las pruebas de sistema también demuestran rendimiento, confiabilidad y evidencian que el sistema no haga aquello que no debe hacer, como producir salidas indeseadas, etcétera. Andrea (2007) sugiere cómo pueden extenderse las herramientas de prueba para integrar algunos aspectos de las pruebas de sistema con TDD.

El desarrollo dirigido por pruebas resulta ser un enfoque exitoso para proyectos de dimensión pequeña y mediana. Por lo general, los programadores que adoptan dicho enfoque están contentos con él y descubren que es una forma más productiva de desarrollar

software (Jeffries y Melnik, 2007). En algunos ensayos, se demostró que conduce a mejorar la calidad del código; en otros, los resultados no son concluyentes. Sin embargo, no hay evidencia de que el TDD conduzca a un código con menor calidad.

8.3 Pruebas de versión

Las pruebas de versión son el proceso de poner a prueba una versión particular de un sistema que se pretende usar fuera del equipo de desarrollo. Por lo general, la versión del sistema es para clientes y usuarios. No obstante, en un proyecto complejo, la versión podría ser para otros equipos que desarrollan sistemas relacionados. Para productos de software, la versión sería para el gerente de producto, quien después la prepara para su venta.

Existen dos distinciones importantes entre las pruebas de versión y las pruebas del sistema durante el proceso de desarrollo:

1. Un equipo independiente que no intervino en el desarrollo del sistema debe ser el responsable de las pruebas de versión.
2. Las pruebas del sistema por parte del equipo de desarrollo deben enfocarse en el descubrimiento de bugs en el sistema (pruebas de defecto). El objetivo de las pruebas de versión es comprobar que el sistema cumpla con los requerimientos y sea suficientemente bueno para uso externo (pruebas de validación).

La principal meta del proceso de pruebas de versión es convencer al proveedor del sistema de que éste es suficientemente apto para su uso. Si es así, puede liberarse como un producto o entregarse al cliente. Por lo tanto, las pruebas de versión deben mostrar que el sistema entrega su funcionalidad, rendimiento y confiabilidad especificados, y que no falla durante el uso normal. Deben considerarse todos los requerimientos del sistema, no sólo los de los usuarios finales del sistema.

Las pruebas de versión, por lo regular, son un proceso de prueba de caja negra, donde las pruebas se derivan a partir de la especificación del sistema. El sistema se trata como una caja negra cuyo comportamiento sólo puede determinarse por el estudio de entradas y salidas relacionadas. Otro nombre para esto es “prueba funcional”, llamada así porque al examinador sólo le preocupa la funcionalidad y no la aplicación del software.

8.3.1 Pruebas basadas en requerimientos

Un principio general de buena práctica en la ingeniería de requerimientos es que éstos deben ser comprobables; esto es, los requerimientos tienen que escribirse de forma que pueda diseñarse una prueba para dicho requerimiento. Luego, un examinador comprueba que el requerimiento se cumpla. En consecuencia, las pruebas basadas en requerimientos son un enfoque sistemático al diseño de casos de prueba, donde se considera cada requerimiento y se deriva un conjunto de pruebas para éste. Las pruebas basadas en requerimientos son pruebas de validación más que de defecto: se intenta demostrar que el sistema implementó adecuadamente sus requerimientos.

Por ejemplo, considere los requerimientos relacionados para el MHC-PMS (presentado en el capítulo 1), que se enfocan a la comprobación de alergias a medicamentos:

Si se sabe que un paciente es alérgico a algún fármaco en particular, entonces la prescripción de dicho medicamento dará como resultado un mensaje de advertencia que se emitirá al usuario del sistema.

Si quien prescribe ignora una advertencia de alergia, deberá proporcionar una razón para ello.

Para comprobar si estos requerimientos se cumplen, tal vez necesite elaborar muchas pruebas relacionadas:

1. Configurar un registro de un paciente sin alergias conocidas. Prescribir medicamentos para alergias que se sabe que existen. Comprobar que el sistema no emite un mensaje de advertencia.
2. Realizar un registro de un paciente con una alergia conocida. Prescribir el medicamento al que es alérgico y comprobar que el sistema emite la advertencia.
3. Elaborar un registro de un paciente donde se reporten alergias a dos o más medicamentos. Prescribir dichos medicamentos por separado y comprobar que se emite la advertencia correcta para cada medicamento.
4. Prescribir dos medicamentos a los que sea alérgico el paciente. Comprobar que se emiten correctamente dos advertencias.
5. Prescribir un medicamento que emite una advertencia y pasar por alto dicha advertencia. Comprobar que el sistema solicita al usuario proporcionar información que explique por qué pasó por alto la advertencia.

A partir de esto se puede ver que probar un requerimiento no sólo significa escribir una prueba. Por lo general, usted deberá escribir muchas pruebas para garantizar que cubrió los requerimientos. También hay que mantener el rastreo de los registros de sus pruebas basadas en requerimientos, que vinculan las pruebas con los requerimientos específicos que se ponen a prueba.

8.3.2 Pruebas de escenario

Las pruebas de escenario son un enfoque a las pruebas de versión donde se crean escenarios típicos de uso y se les utiliza en el desarrollo de casos de prueba para el sistema. Un escenario es una historia que describe una forma en que puede usarse el sistema. Los escenarios deben ser realistas, y los usuarios reales del sistema tienen que relacionarse con ellos. Si usted empleó escenarios como parte del proceso de ingeniería de requerimientos (descritos en el capítulo 4), entonces podría reutilizarlos como escenarios de prueba.

En un breve ensayo sobre las pruebas de escenario, Kaner (2003) sugiere que una prueba de escenario debe ser una historia narrativa que sea creíble y bastante compleja. Tiene que motivar a los participantes; esto es, deben relacionarse con el escenario y creer que es importante que el sistema pase la prueba. También sugiere que debe ser fácil de

Kate es enfermera con especialidad en atención a la salud mental. Una de sus responsabilidades es visitar a domicilio a los pacientes, para comprobar la efectividad de su tratamiento y que no sufran de efectos colaterales del fármaco.

En un día de visitas domésticas, Kate ingresa al MHC-PMS y lo usa para imprimir su agenda de visitas domiciliarias para ese día, junto con información resumida sobre los pacientes por visitar. Solicita que los registros para dichos pacientes se descarguen a su laptop. Se le pide la palabra clave para cifrar los registros en la laptop.

Uno de los pacientes a quienes visita es Jim, quien es tratado con medicamentos antidepresivos. Jim siente que el medicamento le ayuda, pero considera que el efecto colateral es que se mantiene despierto durante la noche. Kate observa el registro de Jim y se le pide la palabra clave para descifrar el registro. Comprueba el medicamento prescrito y consulta sus efectos colaterales. El insomnio es un efecto colateral conocido, así que anota el problema en el registro de Jim y sugiere que visite la clínica para que cambien el medicamento. Él está de acuerdo, así que Kate ingresa un recordatorio para llamarlo en cuanto ella regrese a la clínica, para concertarle una cita con un médico. Termina la consulta y el sistema vuelve a cifrar el registro de Jim.

Más tarde, al terminar sus consultas, Kate regresa a la clínica y sube los registros de los pacientes visitados a la base de datos. El sistema genera para Kate una lista de aquellos pacientes con quienes debe comunicarse, para obtener información de seguimiento y concertar citas en la clínica.

Figura 8.10 Escenario de uso para el MHC-PMS

evaluar. Si hay problemas con el sistema, entonces el equipo de pruebas de versión tiene que reconocerlos. Como ejemplo de un posible escenario para el MHC-PMS, la figura 8.10 describe una forma de utilizar el sistema en una visita domiciliaria, que pone a prueba algunas características del MHC-PMS:

1. Autenticación al ingresar al sistema.
2. Descarga y carga registros de paciente específicos desde una laptop.
3. Agenda de visitas a domicilio.
4. Cifrado y descifrado de registros de pacientes en un dispositivo móvil.
5. Recuperación y modificación de registros.
6. Vinculación con la base de datos de medicamentos que mantenga información acerca de efectos colaterales.
7. Sistema para recordatorio de llamadas.

Si usted es examinador de versión, opere a través de este escenario, interprete el papel de Kate y observe cómo se comporta el sistema en respuesta a las diferentes entradas. Como "Kate", usted puede cometer errores deliberados, como ingresar la palabra clave equivocada para decodificar registros. Esto comprueba la respuesta del sistema ante los errores. Tiene que anotar cuidadosamente cualquier problema que surja, incluidos problemas de rendimiento. Si un sistema es muy lento, esto cambiará la forma en que se usa. Por ejemplo, si se tarda mucho al cifrar un registro, entonces los usuarios que tengan poco tiempo pueden saltar esta etapa. Si pierden su laptop, una persona no autorizada podría ver entonces los registros de los pacientes.

Cuando se usa un enfoque basado en escenarios, se ponen a prueba por lo general varios requerimientos dentro del mismo escenario. Por lo tanto, además de comprobar

requerimientos individuales, también demuestra que las combinaciones de requerimientos no causan problemas.

8.3.3 Pruebas de rendimiento

Una vez integrado completamente el sistema, es posible probar propiedades emergentes, como el rendimiento y la confiabilidad. Las pruebas de rendimiento deben diseñarse para garantizar que el sistema procese su carga pretendida. Generalmente, esto implica efectuar una serie de pruebas donde se aumenta la carga, hasta que el rendimiento del sistema se vuelve inaceptable.

Como con otros tipos de pruebas, las pruebas de rendimiento se preocupan tanto por demostrar que el sistema cumple con sus requerimientos, como por descubrir problemas y defectos en el sistema. Para probar si los requerimientos de rendimiento se logran, quizá se deba construir un perfil operativo. Un perfil operativo (capítulo 15) es un conjunto de pruebas que reflejan la mezcla real de trabajo que manejará el sistema. Por consiguiente, si el 90% de las transacciones en un sistema son del tipo A, el 5% del tipo B, y el resto de los tipos C, D y E, entonces habrá que diseñar el perfil operativo de modo que la gran mayoría de pruebas sean del tipo A. De otra manera, no se obtendrá una prueba precisa del rendimiento operativo del sistema.

Desde luego, este enfoque no necesariamente es el mejor para pruebas de defecto. La experiencia demuestra que una forma efectiva de descubrir defectos es diseñar pruebas sobre los límites del sistema. En las pruebas de rendimiento, significa estresar el sistema al hacer demandas que estén fuera de los límites de diseño del software. Esto se conoce como "prueba de esfuerzo". Por ejemplo, digamos que usted prueba un sistema de procesamiento de transacciones que se diseña para procesar hasta 300 transacciones por segundo. Comienza por probar el sistema con menos de 300 transacciones por segundo. Luego aumenta gradualmente la carga del sistema más allá de 300 transacciones por segundo, hasta que está muy por arriba de la carga máxima de diseño del sistema y el sistema falla. Este tipo de pruebas tiene dos funciones:

1. Prueba el comportamiento de falla del sistema. Pueden surgir circunstancias a través de una combinación inesperada de eventos donde la carga colocada en el sistema supere la carga máxima anticipada. Ante tales circunstancias, es importante que la falla del sistema no cause corrupción de datos o pérdida inesperada de servicios al usuario. Las pruebas de esfuerzo demuestran que la sobrecarga del sistema hace que "falle poco" en vez de colapsar bajo su carga.

2. Fuerza al sistema y puede hacer que salgan a la luz defectos que no se descubrirían normalmente. Aunque se puede argumentar que esos defectos probablemente no causen fallas en el sistema en uso normal, pudiera haber una serie de combinaciones inusuales de circunstancias normales que requieren pruebas de esfuerzo.

Las pruebas de esfuerzo son particularmente relevantes para los sistemas distribuidos basados en redes de procesadores. Dichos sistemas muestran con frecuencia degradación severa cuando se cargan en exceso. La red se empantana con la coordinación de datos que deben intercambiar los diferentes procesos. Éstos se vuelven cada vez más lentos conforme esperan los datos requeridos de otros procesos. Las pruebas de esfuerzo ayudan a descubrir cuándo comienza la degradación, de manera que se puedan adicionar comprobaciones al sistema para rechazar transacciones más allá de este punto.

8.4 Pruebas de usuario

Las pruebas de usuario o del cliente son una etapa en el proceso de pruebas donde los usuarios o clientes proporcionan entrada y asesoría sobre las pruebas del sistema. Esto puede implicar probar de manera formal un sistema que se comisionó a un proveedor externo, o podría ser un proceso informal donde los usuarios experimentan con un nuevo producto de software, para ver si les gusta y si hace lo que necesitan. Las pruebas de usuario son esenciales, aun cuando se hayan realizado pruebas abarcadoras de sistema y de versión. La razón de esto es que la influencia del entorno de trabajo del usuario tiene un gran efecto sobre la fiabilidad, el rendimiento, el uso y la robustez de un sistema.

Es casi imposible que un desarrollador de sistema replique el entorno de trabajo del sistema, pues las pruebas en el entorno del desarrollador forzosamente son artificiales. Por ejemplo, un sistema que se pretenda usar en un hospital se usa en un entorno clínico donde suceden otros hechos, como emergencias de pacientes, conversaciones con familiares del paciente, etcétera. Todo ello afecta el uso de un sistema, pero los desarrolladores no pueden incluirlos en su entorno de pruebas.

En la práctica, hay tres diferentes tipos de pruebas de usuario:

1. Pruebas alfa, donde los usuarios del software trabajan con el equipo de diseño para probar el software en el sitio del desarrollador.
2. Pruebas beta, donde una versión del software se pone a disposición de los usuarios, para permitirles experimentar y descubrir problemas que encuentran con los desarrolladores del sistema.
3. Pruebas de aceptación, donde los clientes prueban un sistema para decidir si está o no listo para ser aceptado por los desarrolladores del sistema y desplegado en el entorno del cliente.

En las pruebas alfa, los usuarios y desarrolladores trabajan en conjunto para probar un sistema a medida que se desarrolla. Esto significa que los usuarios pueden identificar problemas y conflictos que no son fácilmente aparentes para el equipo de prueba de desarrollo. Los desarrolladores en realidad sólo pueden trabajar a partir de los requerimientos, pero con frecuencia esto no refleja otros factores que afectan el uso práctico del software. Por lo tanto, los usuarios brindan información sobre la práctica que ayuda con el diseño de pruebas más realistas.

Las pruebas alfa se usan a menudo cuando se desarrollan productos de software que se venden como sistemas empaquetados. Los usuarios de dichos productos quizás estén satisfechos de intervenir en el proceso de pruebas alfa porque esto les da información oportuna acerca de las características del nuevo sistema que pueden explotar. También reduce el riesgo de que cambios no anticipados al software tengan efectos perturbadores para su negocio. Sin embargo, las pruebas alfa también se utilizan cuando se desarrolla software personalizado. Los métodos ágiles, como XP, abogan por la inclusión del usuario en el proceso de desarrollo y que los usuarios tengan un papel activo en el diseño de pruebas para el sistema.

Las pruebas beta tienen lugar cuando una versión temprana de un sistema de software, en ocasiones sin terminar, se pone a disposición de clientes y usuarios para evaluación.

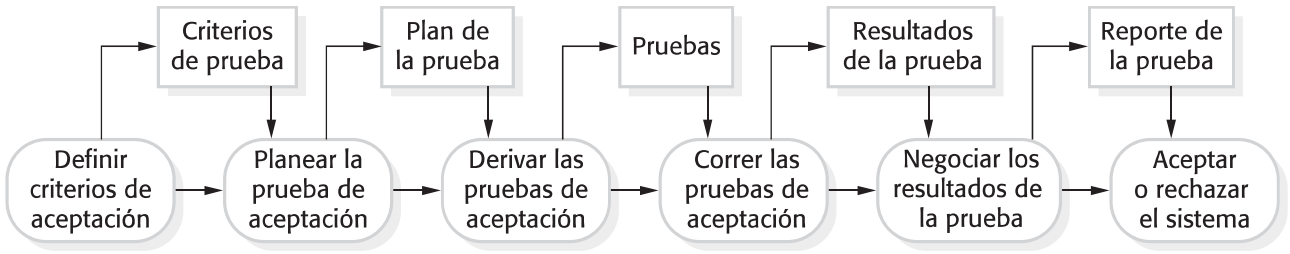


Figura 8.11 Proceso de prueba de aceptación

Los examinadores beta pueden ser un grupo selecto de clientes que sean adoptadores tempranos del sistema. De manera alternativa, el software se pone a disposición pública para uso de quienquiera que esté interesado en él. Las pruebas beta se usan sobre todo para productos de software que se emplean en entornos múltiples y diferentes (en oposición a los sistemas personalizados, que se utilizan por lo general en un entorno definido). Es imposible que los desarrolladores de producto conozcan y repliquen todos los entornos donde se usará el software. En consecuencia, las pruebas beta son esenciales para descubrir problemas de interacción entre el software y las características del entorno donde se emplea. Las pruebas beta también son una forma de comercialización: los clientes aprenden sobre su sistema y lo que puede hacer por ellos.

Las pruebas de aceptación son una parte inherente del desarrollo de sistemas personalizados. Tienen lugar después de las pruebas de versión. Implican a un cliente que prueba de manera formal un sistema, para decidir si debe o no aceptarlo del desarrollador del sistema. La aceptación implica que debe realizarse el pago por el sistema.

Existen seis etapas en el proceso de pruebas de aceptación, como se muestra en la figura 8.11. Éstas son:

1. *Definir los criterios de aceptación* Esta etapa debe, de manera ideal, anticiparse en el proceso, antes de firmar el contrato por el sistema. Los criterios de aceptación forman parte del contrato del sistema y tienen que convenirse entre el cliente y el desarrollador. Sin embargo, en la práctica suele ser difícil definir los criterios de manera tan anticipada en el proceso. Es posible que no estén disponibles requerimientos detallados y que haya cambios significativos en los requerimientos durante el proceso de desarrollo.

2. *Plan de pruebas de aceptación* Esto incluye decidir sobre los recursos, el tiempo y el presupuesto para las pruebas de aceptación, así como establecer un calendario de pruebas. El plan de pruebas de aceptación debe incluir también la cobertura requerida de los requerimientos y el orden en que se prueban las características del sistema. Tiene que definir riesgos al proceso de prueba, como caídas del sistema y rendimiento inadecuado, y resolver cómo mitigar dichos riesgos.

3. *Derivar pruebas de aceptación* Una vez establecidos los criterios de aceptación, tienen que diseñarse pruebas para comprobar si un sistema es aceptable o no. Las pruebas de aceptación deben dirigirse a probar tanto las características funcionales como las no funcionales del sistema (por ejemplo, el rendimiento). Lo ideal sería que dieran cobertura completa a los requerimientos del sistema. En la práctica, es difícil establecer criterios de aceptación completamente objetivos. Con frecuencia hay espacio para argumentar sobre si las pruebas deben mostrar o no que un criterio se cubre de manera definitiva.

4. *Correr pruebas de aceptación* Las pruebas de aceptación acordadas se ejecutan sobre el sistema. De manera ideal, esto debería ocurrir en el entorno real donde se usará el sistema, pero esto podría ser perturbador y poco práctico. En consecuencia, quizá deba establecerse un entorno de pruebas de usuario para efectuar dichas pruebas. Es difícil automatizar este proceso, ya que parte de las pruebas de aceptación podría necesitar poner a prueba las interacciones entre usuarios finales y el sistema. Es posible que se requiera cierta capacitación de los usuarios finales.

5. *Negociar los resultados de las pruebas* Es poco probable que se pasen todas las pruebas de aceptación definidas y que no haya problemas con el sistema. Si éste es el caso, entonces las pruebas de aceptación están completas y el sistema está listo para entregarse. Con mayor regularidad se descubrirán algunos problemas. En tales casos, el desarrollador y el cliente tienen que negociar para decidir si el sistema es suficientemente adecuado para ponerse en uso. También deben acordar sobre la respuesta del desarrollador para identificar problemas.

6. *Rechazo/aceptación del sistema* Esta etapa incluye una reunión entre los desarrolladores y el cliente para decidir si el sistema debe aceptarse o no. Si el sistema no es suficientemente bueno para usarse, entonces se requiere mayor desarrollo para corregir los problemas identificados. Una vez completo, se repite la fase de pruebas de aceptación.

En los métodos ágiles, como XP, las pruebas de aceptación tienen un significado un tanto diferente. En principio, comparten la noción de que son los usuarios quienes deciden si el sistema es aceptable o no. Sin embargo, en XP, el usuario forma parte del equipo de desarrollo (es decir, es un examinador alfa) y proporciona los requerimientos del sistema en términos de historias de usuario. También es responsable de definir las pruebas, que permiten determinar si el software desarrollado soporta o no la historia del usuario. Las pruebas son automatizadas y el desarrollo no avanza sino hasta que se pasan las pruebas de aceptación históricas. Por consiguiente, no hay una actividad separada de pruebas de aceptación.

Como se estudió en el capítulo 3, un problema con la participación del usuario es garantizar que quien se inserte en el equipo de desarrollo sea un usuario "típico" con conocimiento general de cómo se usará el sistema. Quizá sea difícil encontrar a tal usuario y, por lo tanto, las pruebas de aceptación en realidad tal vez no sean un verdadero reflejo de la práctica. Más aún, el requerimiento de pruebas automatizadas limita severamente la flexibilidad de los sistemas interactivos de pruebas. Para tales sistemas, las pruebas de aceptación podrían requerir que grupos de usuarios finales usen el sistema como si fuera parte de su trabajo cotidiano.

Usted puede considerar que las pruebas de aceptación son un conflicto contractual tajante. Si un sistema no pasa sus pruebas de aceptación, debe rechazarse y el pago no se realiza. Sin embargo, la realidad es más compleja. Los clientes quieren usar el software tan pronto como puedan debido a los beneficios de su despliegue inmediato. Ellos quizá compraron un nuevo hardware, capacitaron al personal y modificaron sus procesos. Tal vez están deseosos de aceptar el software, sin importar los problemas, ya que los costos por no usar el software serían mayores que los de trabajar en torno a los problemas. Por consiguiente, el resultado de las negociaciones podría ser la aceptación condicional del sistema. El cliente acepta tal sistema para comenzar el despliegue. El proveedor del sistema acuerda reparar los problemas urgentes y entregar una nueva versión al cliente tan rápido como sea posible.

PUNTOS CLAVE

- Las pruebas sólo pueden mostrar la presencia de errores en un programa. Si embargo, no pueden garantizar que no surjan fallas posteriores.
- Las pruebas de desarrollo son responsabilidad del equipo de desarrollo del software. Un equipo independiente debe responsabilizarse de probar un sistema antes de darlo a conocer a los clientes. En el proceso de pruebas de usuario, clientes o usuarios del sistema brindan datos de prueba y verifican que las pruebas sean exitosas.
- Las pruebas de desarrollo incluyen pruebas de unidad, donde se examinan objetos y métodos individuales; pruebas de componente, donde se estudian grupos de objetos relacionados; y pruebas del sistema, donde se analizan sistemas parciales o completos.
- Cuando pruebe software, debe tratar de "romperlo" mediante la experiencia y los lineamientos que elijan los tipos de casos de prueba que hayan sido efectivos para descubrir defectos en otros sistemas.
- Siempre que sea posible, se deben escribir pruebas automatizadas. Las pruebas se incrustan en un programa que puede correrse cada vez que se hace un cambio al sistema.
- El desarrollo de la primera prueba es un enfoque de desarrollo, donde las pruebas se escriben antes de que se pruebe el código. Se realizan pequeños cambios en el código, y éste se refactoriza hasta que todas las pruebas se ejecuten exitosamente.
- Las pruebas de escenario son útiles porque imitan el uso práctico del sistema. Implican trazar un escenario de uso típico y utilizarlo para derivar casos de prueba.
- Las pruebas de aceptación son un proceso de prueba de usuario, donde la meta es decidir si el software es suficientemente adecuado para desplegarse y utilizarse en su entorno operacional.

LECTURAS SUGERIDAS

"How to design practical test cases". Un artículo práctico sobre el diseño de casos de prueba elaborado por un publicista de una compañía japonesa, que tiene una muy buena reputación debido a que entrega el software con muy pocas fallas. (T. Yamaura, *IEEE Software*, **15** (6), noviembre 1998.) http://dx.doi.org/10.1109/52.730835.

How to Break Software: A Practical Guide to Testing. Se trata de un libro más práctico que teórico, sobre las pruebas de software. El autor presenta un conjunto de lineamientos basados en su experiencia relativa al diseño de pruebas, que probablemente sean efectivas en la detección de fallas del sistema. (J. A. Whittaker, Addison-Wesley, 2002.)

"Software Testing and Verification". Este número especial del *IBM Systems Journal* comprende algunos ensayos de pruebas, incluido un buen panorama. Además, incluye ensayos de métricas de prueba y automatización de pruebas. (*IBM Systems Journal*, **41** (1), enero 2002.)

"Test-driven development". Este número especial es acerca del desarrollo dirigido por pruebas, el cual incluye un buen panorama general del TDD, así como ensayos de experiencia sobre cómo se usó el TDD para diferentes tipos de software. (*IEEE Software*, **24** (3) mayo/junio 2007.)

EJERCICIOS

8.1. Explique por qué no es necesario que un programa esté completamente libre de defectos antes de entregarse a sus clientes.

8.2. Indique por qué las pruebas sólo pueden detectar la presencia de errores, pero no su ausencia.

8.3. Algunas personas argumentan que los desarrolladores no deben intervenir en las pruebas de su propio código, sino que todas las pruebas deben ser responsabilidad de un equipo independiente. Exponga argumentos en favor y en contra de las pruebas efectuadas por parte de los mismos desarrolladores.

8.4. Se pide al lector poner a prueba un método llamado "catWhiteSpace" en un objeto "Paragraph" que, dentro del párrafo, sustituye secuencias de caracteres blancos con un solo carácter blanco. Identifique las particiones de prueba para este ejemplo y derive un conjunto de pruebas para el método "catWhiteSpace".

8.5. ¿Qué es la prueba de regresión? Explique cómo el uso de pruebas automatizadas y un marco de pruebas como JUnit simplifican las pruebas de regresión.

8.6. El MHC-PMS se construyó al adaptar un sistema de información comercial. ¿Cuáles considera que son las diferencias entre probar tal sistema y probar el software que se desarrolló usando un lenguaje orientado a objetos como Java?

8.7. Diseñe un escenario que pueda usar para ayudarse a elaborar pruebas para el sistema de estación meteorológica en campo abierto.

8.8. ¿Qué entiende por "pruebas de esfuerzo"? Sugiera cómo puede hacer una prueba de esfuerzo del MHC-PMS.

8.9. ¿Cuáles son los beneficios de hacer participar a usuarios en las pruebas de versión en una etapa temprana del proceso de pruebas? ¿Hay desventajas en la implicación del usuario?

8.10. Un enfoque común a las pruebas del sistema es probar el sistema hasta que se agote el presupuesto de pruebas y, luego, entregar el sistema a los clientes. Discuta la ética de este enfoque para sistemas que se entregan a clientes externos.

REFERENCIAS

Andrea, J. (2007). "Envisioning the Next Generation of Functional Testing Tools". *IEEE Software*, 24 (3), 58–65.

Beck, K. (2002). *Test Driven Development: By Example*. Boston: Addison-Wesley.

Bezier, B. (1990). *Software Testing Techniques, 2nd edition*. New York: Van Nostrand Rheinhold.

Boehm, B. W. (1979). "Software engineering; R & D Trends and defense needs." In *Research Directions in Software Technology*. Wegner, P. (ed.). Cambridge, Mass.: MIT Press. 1–9.

Cusamano, M. y Selby, R. W. (1998). *Microsoft Secrets*. New York: Simon and Shuster.

Dijkstra, E. W., Dahl, O. J. y Hoare, C. A. R. (1972). *Structured Programming*. Londres: Academic Press.

Fagan, M. E. (1986). "Advances in Software Inspections". *IEEE Trans. on Software Eng.*, SE-12 (7), 744–51.

Jeffries, R. y Melnik, G. (2007). "TDD: The Art of Fearless Programming". *IEEE Software*, **24**, 24–30.

Kaner, C. (2003). "The power of 'What If . . .' and nine ways to fuel your imagination: Cem Kaner on scenario testing". *Software Testing and Quality Engineering*, **5** (5), 16–22.

Lutz, R. R. (1993). "Analyzing Software Requirements Errors in Safety-Critical Embedded Systems". RE'93, San Diego, Calif.: IEEE.

Martin, R. C. (2007). "Professionalism and Test-Driven Development". *IEEE Software*, **24** (3), 32–6.

Massol, V. y Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.

Prowell, S. J., Trammell, C. J., Linger, R. C. y Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.

Whittaker, J. W. (2002). *How to Break Software: A Practical Guide to Testing*. Boston: Addison-Wesley.