CAPÍTULO

18 ESTRATEGIAS DE PRUEBA DE SOFTWARE

UNA estrategia de prueba del software integra las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del software. La estrategia proporciona un mapa que describe los pasos que hay que llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar esos pasos, y cuánto esfuerzo, tiempo y recursos se van a requerir. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los datos resultantes.

Una estrategia de prueba del software debe ser suficientemente flexible para promover la creatividad y la adaptabilidad necesarias para adecuar la prueba a todos los grandes sistemas basados en software. Al mismo tiempo, la estrategia debe ser suficientemente rígida para promover un seguimiento razonable de la planificación y la gestión a medida que progresa el proyecto. Shooman [SHO83] trata estas cuestiones:

> En muchos aspectos, la prueba es un proceso individualista, y el número de tipos diferentes de pruebas varía tanto como los diferentes enfoques de desarrollo. Durante muchos años, nuestra Única defensa contra los errores de programación era un cuidadoso diseño y la propia inteligencia del programador. Ahora nos encontramos en una era en la que las técnicas modernas de diseño [y las revisiones técnicas formales] nos ayudan a reducir el número de errores iniciales que se encuentran en el código de forma inherente. De forma similar, los diferentes métodos de prueba están empezando a agruparse en varias filosofías y enfoques diferentes.

VISTAZO RÁPIDO

¿Qué es? El diseño efectivo de casos de prueba (Capítulo 17) es importante, pero también lo es la estrategia para su utilización. ¿Se deberá desarrollar un plan formal para estas pruebas?. ¿Se deberá probar el programa íntegramente o ejecutar pruebas solamente sobre una parte pequeña del mismo? ¿Se deberán ejecutar pruebas de regresión cuando se añadan nuevos componentes al sistema? ¿Cuándo se deberá involucrar al cliente? Estas y otras muchas cuestiones serán contestadas cuando desarrolles una estrategia de prueba del software.

¿Quién lo hace? El jefe del proyecto, los ingenieros del software y los especialistas en pruebas.

¿Por qué es importante? En el proyecto, la prueba a veces requiere más esfuerzo que cualquier otra actividad de la ingeniería del software. Si se efectúa sin un plan, el tiempo se desaprovecha y el esfuerzo es consumido innecesariamente y, en el peor de los casos, los errores inadvertidos quedarán sin detectar. Por tanto, parece razonable establecer una estrategia sistemática para probar el software.

¿Cuáles son los pasos? La prueba comienza por «lo pequeño» y progresa hacia «lo grande». Por esta razón, debemos comenzar las primeras pruebas sobre el componente elemental y aplicar sobre él pruebas de caja blanca y de caja negra para descubrir errores en la lógica y en la funcionalidad del programa. Después de que los componentes elementales hayan sido aprobados, procederemos a su integración. Las pruebas se efectuarán conforme el software se vaya construyendo.

Finalmente, una serie de pruebas de alto nivel serán ejecutadas una vez el programa esté totalmente preparado para su operatividad. Estas pruebas están diseñadas para descubrir errores en los requisitos.

¿Cuál es el producto obtenido? El equipo de trabajo que desarrolla el software documenta la *especificación de la prueba* basándose en la definición del plan que establece la estrategia general y del procedimiento que específicamente describe los pasos a seguir y las pruebas a realizar.

¿Cómo puedo estar seguro de que lo he hecho correctamente? La revisión de la *especificación de la prueba* es previa a la realización de la prueba. Se debe valorar la completitud de los casos de prueba y de las tareas de la prueba. Un plan y un procedimiento de prueba efectivo permitirá una construcción ordenada del software y el descubrimiento de errores en cada etapa del proceso de construcción.

Estas «filosofías y enfoques» constituyen lo que nosotros llamaremos estrategia. En el Capítulo 17 se presentó la tecnología de prueba del software[1]. En este capítulo centraremos nuestra atención en las estrategias de prueba del software.

[1] Las pruebas de sistemas orientados a objetos se estudian en el Capítulo 23.

18.1 UN ENFOQUE ESTRATÉGICO PARA LAS PRUEBAS DEL SOFTWARE

Las pruebas son un conjunto de actividades que se pueden planificar por adelantado y llevar a cabo sistemáticamente. Por esta razón, se debe definir en el proceso de la ingeniería del software una *plantilla* para las pruebas del software: un conjunto de pasos en los que podamos situar los métodos específicos de diseño de casos de prueba.

Se han propuesto varias estrategias de prueba del software en distintos libros. Todas proporcionan al ingeniero del software una plantilla para la prueba y todas tienen las siguientes características generales:

- Las pruebas comienzan a nivel de módulo[2] y trabajan «hacia fuera», hacia la integración de todo el sistema basado en computadora.
- Según el momento, son apropiadas diferentes técnicas de prueba.
- La prueba la lleva a cabo el responsable del desarrollo del software y (para grandes proyectos) un grupo independiente de pruebas.
- La prueba y la depuración son actividades diferentes, pero la depuración se debe incluir en cualquier estrategia de prueba.

Una estrategia de prueba del software debe incluir pruebas de bajo nivel que verifiquen que todos los pequeños segmentos de código fuente se han implementado correctamente, así como pruebas de alto nivel que validen las principales funciones del sistema frente a los requisitos del cliente. Una estrategia debe proporcionar una guía al profesional y proporcionar un conjunto de hitos para el jefe de proyecto. Debido a que los pasos de la estrategia de prueba se dan a la vez cuando aumenta la presión de los plazos fijados, se debe poder medir el progreso y los problemas deben aparecer lo antes posible.

Información útil sobre las estrategias de prueba del software es suministrada en el informe sobre la Prueba del Software en: **www.ondaweb.com/sti/newsltr.htm**

18.1.1. Verificación y validación

La prueba del software es un elemento de un tema más amplio que, a menudo, es conocido como verificación y validación (V&V). La *verificación* se refiere al conjunto de actividades que aseguran que el software implementa correctamente una función específica. La *validación* se refiere a un conjunto diferente de actividades que aseguran que el software construido se ajusta a los requisitos del cliente. Bohem [BOE81] lo define de otra forma:

> *Verificación:* «¿Estamos construyendo el producto correctamente?»
>
> *Validación:* «¿Estamos construyendo el producto correcto?»

La definición de V&V comprende muchas de las actividades a las que nos hemos referido como garantía de calidad del software (SQA*).

La verificación y la validación abarcan una amplia lista de actividades SQA que incluye: revisiones técnicas formales, auditorías de calidad y de configuración, monitorización de rendimientos, simulación, estudios de factibilidad, revisión de la documentación, revisión de la base de datos, análisis algorítmico, pruebas de desarrollo, pruebas de validación y pruebas de instalación [WAL89]. A pesar de que las actividades de prueba tienen un papel muy importante en V&V, muchas otras actividades son también necesarias.

Referencia cruzada

Las actividades SQA son estudiadas en detalle en el Capítulo 8.

Las pruebas constituyen el último bastión desde el que se puede evaluar la calidad y, de forma más pragmática, descubrir los errores. Pero las pruebas no deben ser vistas como una red de seguridad. Como se suele decir: «No se puede probar la calidad. Si no está ahí antes de comenzar la prueba, no estará cuando se termine.» La calidad se incorpora en el software durante el proceso de ingeniería del software. La aplicación adecuada de los métodos y de las herramientas, las revisiones técnicas formales efectivas y una sólida gestión y medición, conducen a la calidad, que se confirma durante las pruebas.

Cita:

La prueba es una parte inevitable de cualquier esfuerzo responsable para desarrollar un sistema software.

William Howden

Miller [MIL77] relaciona la prueba del software con la garantía de calidad al establecer que «la motivación subyacente de la prueba de programas es confirmar la calidad del software con métodos que se pueden aplicar de forma económica y efectiva, tanto a grandes como a pequeños sistemas».

[2] Para los sistemas orientados a objetos, las pruebas empiezan a nivel de clase o de objeto. Vea más detalles en el Capítulo 23.

* Por lo habitual de su utilización mantenemos el termino ingles SQA (Software Quality Assurance).

18.1.2. Organización para las pruebas del software

En cualquier proyecto de software existe un conflicto de intereses inherente que aparece cuando comienzan las pruebas. Se pide a la gente que ha construido el software que lo pruebe. Esto parece totalmente inofensivo: después de todo, ¿quién puede conocer mejor un programa que los que lo han desarrollado? Desgraciadamente, esos mismos programadores tienen un gran interés en demostrar que el programa está libre de errores, que funciona de acuerdo con las especificaciones del cliente y que estará listo de acuerdo con los plazos y el presupuesto. Cada uno de estos intereses se convierte en inconveniente a la hora de encontrar errores a lo largo del proceso de prueba.

Desde un punto de vista psicológico, el análisis y el diseño del software (junto con la codificación) son tareas constructivas. El ingeniero del software crea un programa de computadora, su documentación y sus estructuras de datos asociadas. Al igual que cualquier constructor, el ingeniero del software está orgulloso del edificio que acaba de construir y se enfrenta a cualquiera que intente sacarle defectos.

Cuando comienza la prueba, aparece una sutil, aunque firme intención de «romper» lo que el ingeniero del software ha construido. Desde el punto de vista del constructor, la prueba se puede considerar (psicológicamente) *destructiva*. Por tanto, el constructor anda con cuidado, diseñando y ejecutando pruebas que demuestren que el programa funciona, en lugar de detectar errores. Desgraciadamente, los errores seguirán estando. Y si el ingeniero del software no los encuentra, ¡el cliente **sí** lo hará!

A menudo, existen ciertos malentendidos que se pueden deducir equivocadamente de la anterior discusión: (1) el responsable del desarrollo no debería entrar en el proceso de prueba; (2) el software debe ser «puesto a salvo» de extraños que puedan probarlo de forma despiadada; (3) los encargados de la prueba sólo aparecen en el proyecto cuando comienzan las etapas de prueba. Todas estas frases son incorrectas.

El responsable del desarrollo del software siempre es responsable de probar las unidades individuales (módulos) del programa, asegurándose de que cada una lleva a cabo la función para la que fue diseñada. En muchos casos, también se encargará de la prueba de integración, el paso de las pruebas que lleva a la construcción (y prueba) de la estructura total del sistema. Sólo una vez que la arquitectura del software esté completa entra en juego un grupo independiente de prueba.

PUNTO CLAVE

Un grupo independiente de prueba no tiene el «conflicto de intereses» que tienen los desarrolladores del software.

El papel del grupo independiente de prueba (GIP) es eliminar los inherentes problemas asociados con el hecho de permitir al constructor que pruebe lo que ha construido. Una prueba independiente elimina el conflicto de intereses que, de otro modo, estaría presente. Después de todo, al personal del equipo que forma el grupo independiente se le paga para que encuentre errores.

Sin embargo, el responsable del desarrollado del software no entrega simplemente el programa al GIP y se desentiende. El responsable del desarrollado y el GIP trabajan estrechamente a lo largo del proyecto de software para asegurar que se realizan pruebas exhaustivas. Mientras se realiza la prueba, el desarrollador debe estar disponible para corregir los errores que se van descubriendo.

Si no existe un GIP en tu organización, tendrás que asumir su punto de vista. Cuando pruebes, intenta destrozar el software.

El GIP es parte del equipo del proyecto de desarrollo de software en el sentido de que se ve implicado durante el proceso de especificación y sigue implicado (planificación y especificación de los procedimientos de prueba) a lo largo de un gran proyecto. Sin embargo, en muchos casos, el GIP informa a la organización de garantía de calidad del software, consiguiendo de este modo un grado de independencia que no sería posible si fuera una parte de la organización de desarrollo del software.

18.1.3. Una estrategia de prueba del software

El proceso de ingeniería del software se puede ver como una espiral, como se ilustra en la Figura 18.1. Inicialmente, la ingeniería de sistemas define el papel del software y conduce al análisis de los requisitos del software, donde se establece el dominio de información, la función, el comportamiento, el rendimiento, las restricciones y los criterios de validación del software. Al movemos hacia el interior de la espiral, llegamos al diseño y, por último, a la codificación. Para desarrollar software de computadora, damos vueltas en espiral a través de una serie de flujos o líneas que disminuyen el nivel de abstracción en cada vuelta,

¿Qué es la estrategia global para la prueba del software?

También se puede ver la estrategia para la prueba del software en el contexto de la espiral (Fig. 18.1). La *prueba de unidad* comienza en el vértice de la espiral y se centra en cada unidad del software, tal como está implementada en código fuente. La prueba avanza, al movernos hacia fuera de la espiral, hasta llegar a la *prueba de integración*, donde el foco de atención es el diseño y la construcción de la arquitectura del software. Dando otra

vuelta por la espiral hacia fuera, encontramos la *prueba de validación,* donde se validan los requisitos establecidos como parte del análisis de requisitos del software, comparándolos con el sistema que ha sido construido. Finalmente, llegamos a la *prueba del sistema,* en la que se prueban como un todo el software y otros elementos del sistema. Para probar software de computadora nos movemos hacia fuera por una espiral que, a cada vuelta, aumenta el alcance de la prueba.

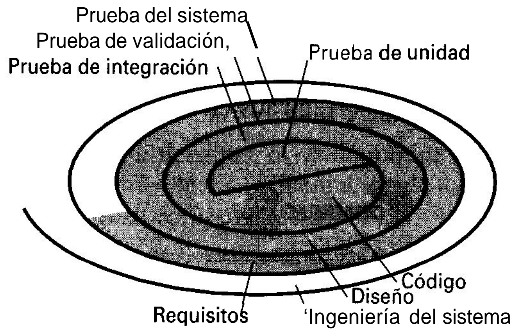


FIGURA 18.1. Estrategia de prueba.

Si consideramos el proceso desde el punto de vista procedimental, la prueba, en el contexto de la ingeniería del software, realmente es una serie de cuatro pasos que se llevan a cabo secuencialmente. Esos pasos se muestran en la Figura 18.2. Inicialmente, la prueba se centra en cada módulo individualmente, asegurando que funcionan adecuadamente como una unidad. De ahí el nombre de *prueba de unidad.* La prueba de unidad hace un uso intensivo de las técnicas de prueba de caja blanca, ejercitando caminos específicos de la estructura de control del módulo para asegurar **un** alcance completo **y** una detección máxima de errores. A continuación, se deben ensamblar o integrar los módulos para formar el paquete de software completo. La *prueba de integración* se dirige a todos los aspectos asociados con el doble problema de verificación **y** de construcción del programa. Durante la integración, las técnicas que más prevalecen son las de diseño de casos de prueba de caja negra, aunque se pueden llevar a cabo algunas pruebas de caja blanca con el fin de asegurar que se cubren los principales caminos de control. Después de que el software se ha integrado (construido), se dirigen un conjunto de *pruebas de alto nivel.* Se deben comprobar los criterios de validación (establecidos durante el análisis de requisitos). La *prueba de validación* proporciona una seguridad final de que el software satisface todos los requisitos funcionales, de comportamiento y de rendimiento. Durante la validación se usan exclusivamente técnicas de prueba de caja negra.

Referencia cruzada

Los técnicas de pruebo de cojo blanca y cojo negra se estudian en el Capítulo 17.

El último paso de prueba de alto nivel queda fuera de los límites de la ingeniería del software, entrando en el más amplio contexto de la ingeniería de sistemas de computadora.

El software, una vez validado, se debe combinar con otros elementos del sistema (por ejemplo, hardware, gente, bases de datos). La *prueba del sistema* verifica que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema total.

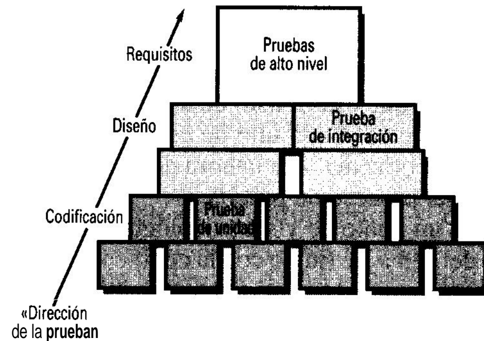


FIGURA 18.2. Etapas en la prueba del software.

18.1.4. Criterios para completar la prueba

Cada vez que se tratan las pruebas del software surge una pregunta clásica: ¿Cuando hemos terminado las pruebas?, ¿cómo sabemos que hemos probado lo suficiente? Desgraciadamente, no hay una respuesta definitiva a esta pregunta, pero hay algunas respuestas prácticas **y** nuevos intentos de base empírica.

¿Cuándo debemos probar?

Una respuesta a la pregunta anterior es: «La prueba nunca termina, ya que el responsable del desarrollo del software carga o pasa el problema al cliente.» Cada vez que el cliente/usuario ejecuta un programa de computadora, dicho programa se está probando con un nuevo conjunto de datos. Este importante hecho subraya la importancia de otras actividades de garantía de calidad del software. Otra respuesta (algo cínica, pero sin embargo cierta) es: «Se termina la prueba cuando se agota el tiempo o el dinero disponible para tal efecto.»

Aunque algunos profesionales se sirvan de estas respuestas como argumento, un ingeniero del software necesita un criterio más riguroso para determinar cuando se ha realizado la prueba suficiente. Musa y Ackerman [MUS89] sugieren una respuesta basada en un criterio estadístico: «No, no podemos tener la absoluta certeza de que el software nunca fallará, pero en base a un modelo estadístico de corte teórico y validado experimentalmente, hemos realizado las pruebas suficientes para decir, con un 95 por 100 de certeza, que la probabilidad de funcionamiento libre de fallo de 1.000 horas de CPU, en un entorno definido de forma probabilística, es al menos 0,995.»

Mediante el modelado estadístico **y** la teoría de fiabilidad del software, se pueden desarrollar modelos de fallos

del software (descubiertos durante las pruebas) como una función del tiempo de ejecución. Una versión del modelo de fallos, denominado *modelo logarítmico de Poisson de tiempo de ejecución,* toma la siguiente forma:

$$f(t) = (1/p) \ln [(l_0 pt + 1)] \qquad (18.1)$$

donde

$f(t)$ número acumulado de fallos que se espera que se produzcan una vez que se ha probado el software durante una cierta cantidad de tiempo de ejecución t,

l_0 la intensidad de fallos inicial del software (fallos por unidad de tiempo) al principio de la prueba

p la reducción exponencial de intensidad de fallo a medida que se encuentran los errores y se van haciendo las correcciones.

La intensidad de fallos instantánea, $l(t)$ se puede obtener mediante la derivada de $f(t)$:

$$l(t) = l_0 / (l_0 pt + 1) \qquad (18.2)$$

Mediante la relación de la ecuación (18.2), los que realizan las pruebas pueden predecir la disminución de errores a medida que estas avanzan. La intensidad de error real se puede trazar junto a la curva predecida (Fig. **18.3**). Si los datos reales recopilados durante la prueba y el modelo logarítmico de Poisson de tiempo de ejecución están razonablemente cerca unos de otros, sobre un número de puntos de datos, el modelo se puede usar para predecir el tiempo de prueba total requerido para alcanzar una intensidad de fallos aceptablemente baja.

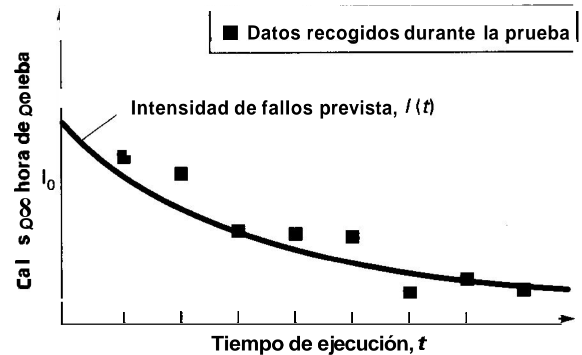


FIGURA 18.3. Intensidad de fallos en función del tiempo de ejecución.

Mediante la agrupación de métricas durante la prueba del software y haciendo uso de los modelos de fiabilidad del software existentes, es posible desarrollar directrices importantes para responder a la pregunta: ¿cuándo terminamos la prueba? No hay duda que todavía queda mucho trabajo por hacer antes de que se puedan establecer reglas cuantitativas para la prueba, pero los enfoques empíricos que existen actualmente son considerablemente mejores que la pura intuición.

18.2 ASPECTOS ESTRATÉGICOS

Más adelante, en este capítulo, exploramos una estrategia sistemática para la prueba del software. Pero incluso la mejor estrategia fracasará si no se tratan una serie de aspectos invalidantes. Tom Gilb [GIL95] plantea que se deben abordar los siguientes puntos si se desea implementar con éxito una estrategia de prueba del software:

¿Qué debemos hacer para definir una estrategia de prueba correcta?

Especificar los requisitos del producto de manera cuantificable mucho antes de que comiencen las pruebas. Aunque el objetivo principal de la prueba es encontrar errores, una buena estrategia de prueba también evalúa otras características de la calidad, tales como la portabilidad, facilidad de mantenimiento y facilidad de uso (Capítulo 19). Todo esto debería especificarse de manera que sea medible para que los resultados de la prueba no sean ambiguos.

Establecer los objetivos de la prueba de manera explícita. Se deberían establecer en términos medibles los objetivos específicos de la prueba. Por ejemplo, la efectividad de la prueba, la cobertura de la prueba, tiempo medio de fallo, el coste para encontrar y arreglar errores, densidad de fallos remanente o frecuencia de ocurrencia, y horas de trabajo por prueba de regresión deberían establecerse dentro de la planificación de la prueba [GIL95].

Comprender qué usuarios van a manejar el software y desarrollar un perfil para cada categoría de usuario. Usar casos que describan el escenario de interacción para cada clase de usuario pudiendo reducir el esfuerzo general de prueba concentrando la prueba en el empleo real del producto.

Referencia cruzada

los casos de usa describen un escenario para usar el software y se estudian en el Capítulo 11.

Desarrollar un plan de prueba que haga hincapié en la «prueba de ciclo rápido». Gilb [GIL95] recomienda que un equipo de ingeniería del software «aprenda a probar en ciclos rápidos (2 por 100 del esfuerzo del proyecto) de incrementos de funcionalidad y/o mejora de la calidad Útiles para el cliente, y que se puedan probar sobre el terreno». La realimentación generada por estas pruebas de ciclo rápido puede usarse para controlar los niveles de calidad y las correspondientes estrategias de prueba.

Cita:

La prueba sólo de los requisitos percibidos por el usuario final es semejante a la inspección de una construcción basada en el trabajo realizado por un decorador sobre el gasto en cimientos, vigas y fontanería.

Boris Beizer

Construir un software «robusto» diseñado para probarse a sí mismo. El software debería diseñarse de manera que use técnicas de depuración antierrores (Sección 18.3.1). **Es** decir, el software debería ser capaz de diagnosticar ciertas clases de errores. Además, el diseño debería incluir pruebas automatizadas y pruebas de regresión.

Usar revisiones técnicas formales efectivas como filtro antes de la prueba. Las revisiones técnicas formales (Capítulo **8**) pueden ser tan efectivas como las pruebas en el descubrimiento de errores. **Por** este motivo, las revisiones pueden reducir la cantidad de esfuerzo de prueba necesaria para producir software de alta calidad.

Llevar a cabo revisiones técnicas formales para evaluar la estrategia de prueba y los propios casos de prueba. Las revisiones técnicas formales pueden descubrir inconsistencias, omisiones y errores claros en el enfoque de la prueba. Esto ahorra tiempo y también mejora la calidad del producto.

Desarrollar un enfoque de mejora continua al proceso de prueba. Debería medirse la estrategia de prueba. Las métricas agrupadas durante la prueba deberían usarse como parte de un enfoque estadístico de control del proceso para la prueba del software.

18.3 PRUEBA DE UNIDAD

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software: el componente software o módulo.

18.3.1. Consideraciones sobre la prueba de unidad

Las pruebas que se dan como parte de la prueba de unidad están esquemáticamente ilustradas en la Figura 18.4. Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa que está siendo probada. Se examinan las estructuras de datos locales para asegurar que los datos que se mantienen temporalmente conservan su integridad durante todos los pasos de ejecución del algoritmo. Se prueban las condiciones límite para asegurar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento. Se ejercitan todos los caminos independientes (caminos básicos) de la estructura de control con el fin de asegurar que todas las sentencias del módulo se ejecutan por lo menos una vez. **Y,** finalmente, se prueban todos los caminos de manejo de errores.

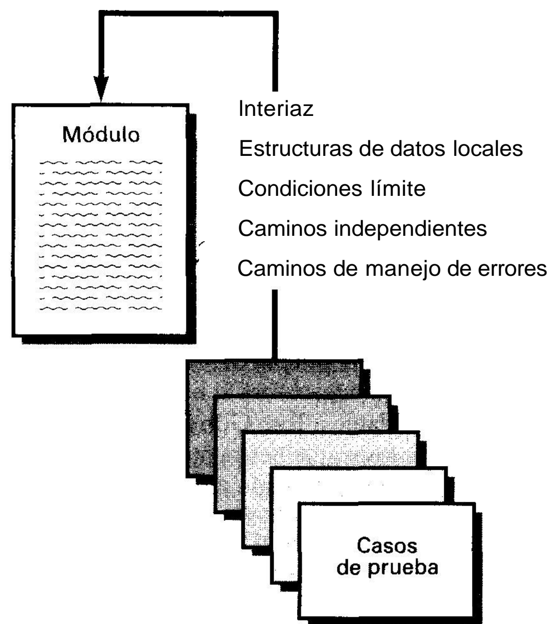
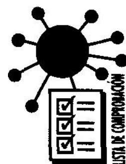


FIGURA 18.4. Prueba de unidad.



Prueba de Unidad.

Antes de iniciar cualquier otra prueba es preciso probar el flujo de datos de la interfaz del módulo. Si los datos no entran correctamente, todas las demás pruebas no tienen sentido. Además de las estructuras de datos locales, durante la prueba de unidad se debe comprobar (en la medida de lo posible) el impacto de los datos globales sobre el módulo.

Durante la prueba de unidad, la comprobación selectiva de los caminos de ejecución es una tarea esencial. Se deben diseñar casos de prueba para detectar errores debidos a cálculos incorrectos, comparaciones incorrectas o flujos de control inapropiados. Las pruebas del camino básico y de bucles son técnicas muy efectivas para descubrir una gran cantidad de errores en los caminos.

¿Qué errores son los más comunes durante la prueba de unidad?

Entre los errores más comunes en los cálculos están: (1) precedencia aritmética incorrecta o mal interpretada;

(2) operaciones de modo mezcladas; (3) inicializaciones incorrectas; (4) falta de precisión; *(5)* incorrecta representación simbólica de una expresión. Las comparaciones y el flujo de control están fuertemente emparejadas (por ejemplo, el flujo de control cambia frecuentemente tras una comparación). Los casos de prueba deben descubrir errores como: (1) comparaciones ente tipos de datos distintos; (2) operadores lógicos o de precedencia incorrectos; (3) igualdad esperada cuando los errores de precisión la hacen poco probable; (4) variables o comparaciones incorrectas; *(5)* terminación de bucles inapropiada o inexistente; (6) fallo de salida cuando se encuentra una iteración divergente, y (7) variables de bucles modificadas de forma inapropiada.

Un buen diseño exige que las condiciones de error sean previstas de antemano y que se dispongan unos caminos de manejo de errores que redirijan o terminen de una forma limpia el proceso cuando se dé un error. Yourdon [YOU75] llama a este enfoque *antipurgado.* Desgraciadamente, existe una tendencia a incorporar la manipulación de errores en el software y así no probarlo nunca. Como ejemplo, sirve una historia real:

Mediante un contrato se desarrolló un importante sistema de diseño interactivo. En un módulo de proceso de transacciones, un bromista puso el siguiente mensaje de manipulación de error que aparecía tras una serie de pruebas condicionales que invocaban varias ramificaciones del flujo de control: ¡ERROR! NO HAY FORMA DE QUE VD. LLEGUE HASTA AQUÍ. ¡Este «mensaje de error» fue descubierto por un cliente durante la fase de puesta a punto!

Asegura que tu diseño de pruebas ejecuta todos los caminos pura encontrar errores. Si no lo haces, el camino puede fallar al ser invocado, provocando una situación incierta.

Entre los errores potenciales que se deben comprobar cuando se evalúa la manipulación de errores están:

1. Descripción ininteligible del error.
2. El error señalado no se corresponde con el error encontrado.
3. La condición de error hace que intervenga el sistema antes que el mecanismo de manejo de errores.
4. El procesamiento de la condición excepcional es incorrecto.
5. La descripción del error no proporciona suficiente información para ayudar en la localización de la causa del error.

La prueba de límites es la última (y probablemente, la más importante) tarea del paso de la prueba de unidad. El software falla frecuentemente en sus condiciones límite. Es decir, con frecuencia aparece un error cuando se procesa el elemento n-ésimo de un array n-dimensional, cuando se hace la i-ésima repetición de un bucle de i pasos o cuando se encuentran los valores máximo o mínimo permitidos. Los casos de prueba que ejerciten las estructuras de datos, el flujo de control y los valores de los datos por debajo y por encima de los máximos y los mínimos son muy apropiados para descubrir estos errores.

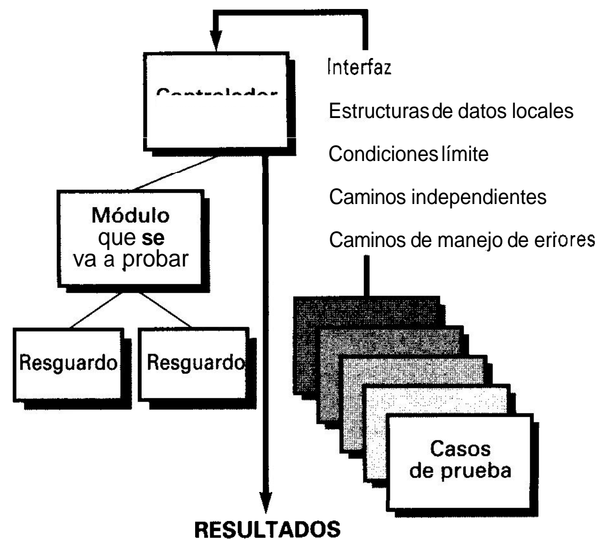


FIGURA 18.5. Entorno para la prueba de unidad.

18.3.2. Procedimientos de Prueba de Unidad

Debido a que un componente no es un programa independiente, se debe desarrollar para cada prueba de unidad un software que controle y/o resguarde. En la Figura 18.5 se ilustra el entorno para la prueba de unidad. En la mayoría de las aplicaciones, un *controlador* no es más que un «programa principal» que acepta los datos del caso de prueba, pasa estos datos al módulo (a ser probado) e imprime los resultados importantes. Los resguardos sirven para reemplazar módulos que están subordinados (llamados por) el componente que hay que probar. Un resguardo o un «subprograma simulado» usa la interfaz del módulo subordinado, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y devuelve control al módulo de prueba que lo invocó.

Hay ocasiones en que no dispones de los recursos para hacer una prueba unitaria completa. En esta situación, selecciona los módulos críticos y aquellos con alta complejidad ciclomática y realiza sobre ellos la prueba unitaria.

Los controladores y los resguardos son una sobrecarga de trabajo. Es decir, ambos son software que debe desarrollarse (normalmente no se aplica un diseño formal) pero que no se entrega con el producto de software final. Si los controladores y resguardos son sencillos, el trabajo adicional es relativamente pequeño. Desgraciadamente, muchos componentes no pue-

den tener una adecuada prueba unitaria con un «sencillo» software adicional. En tales casos, la prueba completa se pospone hasta que se llegue al paso de prueba de integración (donde también se usan controladores o resguardos).

La prueba de unidad se simplifica cuando se diseña un módulo con un alto grado de cohesión. Cuando un módulo sólo realiza una función, se reduce el número de casos de prueba y los errores se pueden predecir y descubrir más fácilmente.

18.4 PRUEBA DE INTEGRACIÓN[3]

Un neófito del mundo del software podría, una vez que se les ha hecho la prueba de unidad a todos los módulos, cuestionar de forma aparentemente legítima lo siguiente: «Si todos funcionan bien por separado, ¿por **qué** dudar de que funcionen todos juntos?» Por supuesto, el problema es «ponerlos juntos» (interacción). Los datos se pueden perder en una interfaz; un módulo puede tener un efecto adverso e inadvertido sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptada individualmente puede crecer hasta niveles inaceptables; y las estructuras de datos globales pueden presentar problemas. Desgraciadamente, la lista sigue y sigue.

La prueba de integración es una técnica sistemática para construir la estructura del programa mientras que, al mismo tiempo, se llevan a cabo pruebas para detectar errores asociados con la interacción. El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.

Efectuar una integración big bag es una estrategia vaga que está condenada al fracaso. la prueba de integración deberá ser conducida incrementalmente.

A menudo hay una tendencia a intentar una integración no incremental; es decir, a construir el programa mediante un enfoque de *«big bang»*. Se combinan todos los módulos por anticipado. Se prueba todo el programa en conjunto. ¡Normalmente se llega al caos! Se encuentran un gran conjunto de errores. La corrección se hace difícil, puesto que es complicado aislar las causas al tener delante el programa entero en toda su extensión. Una vez que se corrigen esos errores aparecen otros nuevos y el proceso continúa en lo que parece ser un ciclo sin fin.

La integración incremental es la antítesis del enfoque del «big bang». El programa se construye y se prueba en pequeños segmentos en los que los errores son más fáciles de aislar y de corregir, es más probable que se puedan probar completamente las interfaces y se puede aplicar un enfoque de prueba sistemática. En las siguientes secciones se tratan varias estrategias de integración incremental diferentes.

18.4.1. Integración descendente

La prueba de integración descendente es un planteamiento incremental a la construcción de la estructura de programas. Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal (programa principal). Los módulos subordinados (subordinados de cualquier modo) al módulo de control principal se van incorporando en la estructura, bien de forma *primero-en-profundidad*, o bien de forma *primero-en-anchura.*

Cuando desarrollas una planificación detallada del proyecto debes considerar la manera en que la integración se va a realizar, de forma que los componentes estén disponibles cuando se necesiten.

Como se muestra en la Figura 18.6, la integración primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura. La selección del camino principal es, de alguna manera, arbitraria y dependerá de las características específicas de la aplicación. Por ejemplo, si se elige el camino de la izquierda, se integrarán primero los módulos M1, M2 y M5. **A** continuación, se integrará M8 o M6 (si es necesario para un funcionamiento adecuado de M2). Acto seguido se construyen los caminos de control central y derecho. La integración primero-en-anchura incorpora todos los módulos directamente subordinados a cada nivel, moviéndose por la estructura de forma horizontal. Según la figura, los primeros módulos que se integran son M2, M3 y **M4.** A continuación, sigue el siguiente nivel de control, M5, M6, etc.

¿ Cuáles son los pasos para una integración top-down?

El proceso de integración se realiza en una serie de cinco pasos:

[3] Las estrategias de integración para sistemas orientados a objetos se tratan en el Capitulo 23.

1. Se usa el módulo de control principal como controlador de la prueba, disponiendo de resguardos para todos los módulos directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración elegido (es decir, primero-en-profundidad o primero-en-anchura) se van sustituyendo uno a uno los resguardos subordinados por los módulos reales.
3. Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
4. Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.
5. *Se* hace la prueba de regresión (Sección **18.4.3**) para asegurarse de que no se han introducido errores nuevos.

El proceso continúa desde el paso 2 hasta que se haya construido la estructura del programa entero.

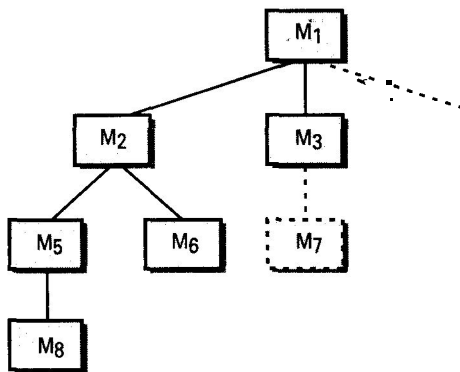


FIGURA 18.6. Integración descendente,

La estrategia de integración descendente verifica los puntos de decisión o de control principales al principio del proceso de prueba. En una estructura de programa bien fabricada, la toma de decisiones se da en los niveles superiores de la jerarquía y, por tanto, se encuentran antes. Si existen problemas generales de control, es esencial reconocerlos cuanto antes. Si se selecciona la integración primero en profundidad, se puede ir implementando y demostrando las funciones completas del software. Por ejemplo, considere una estructura clásica de transacción (Capítulo **14**) en la que se requiere una compleja serie de entradas interactivas, obtenidas y validadas por medio de un camino de entrada. Ese camino de entrada puede ser integrado en forma descendente. Así, se puede demostrar todo el proceso de entradas (para posteriores operaciones de transacción) antes de que se integren otros elementos de la estructura. La demostración anticipada de las posibilidades funcionales es un generador de confianza tanto para el desanollador como para el cliente.

Referencia cruzada

lo fabricación es importante *para* ciertos estilos de *arquitectura*. Para **más** detalles ver el Capítulo 14..

La estrategia descendente parece relativamente fácil, pero, en la práctica, pueden surgir algunos problemas logísticos. El más común de estos problemas se da cuando se requiere un proceso de los niveles más bajos de la jerarquía para poder probar adecuadamente los niveles superiores. Al principio de la prueba descendente, los módulos de bajo nivel se reemplazan por ***resguardos;*** por tanto, no pueden fluir datos significativos hacia arriba por la estructura del programa. El responsable de la prueba tiene tres opciones: (1) retrasar muchas de las pruebas hasta que los resguardos sean reemplazados por los módulos reales; (2) desarrollar resguardos que realicen funciones limitadas que simulen los módulos reales; o *(3)* integrar el software desde el fondo de la jerarquía hacia arriba.

¿Qué problemas puedes encontrar cuando elijas una estrategia de integración descendente?

El primer enfoque (retrasar pruebas hasta reemplazar los resguardos por los módulos reales) hace que perdamos cierto control sobre la correspondencia de ciertas pruebas específicas con la incorporación de determinados módulos. Esto puede dificultar la determinación de las causas del error y tiende a violar la naturaleza altamente restrictiva del enfoque descendente. El segundo enfoque es factible pero puede llevar a un significativo incremento del esfuerzo a medida que los resguardos se hagan más complejos. El tercer enfoque, denominado prueba ascendente, se estudia en la siguiente sección.

18.4.2. Integración ascendente

La prueba de la integración ascendente, como su nombre indica, empieza la construcción y la prueba con los *módulos atómicos* (es decir, módulos de los niveles más bajos de la estructura del programa). Dado que los módulos se integran de abajo hacia arriba, el proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos.

¿Cuáles son los pasos para una integración ascendente?

Se puede implementar una estrategia de integración ascendente mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en ***grupos*** (a veces denominados construcciones) que realicen una subfunción específica del software.
2. Se escribe un controlador (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba.
3. Se prueba el grupo.
4. Se eliminan los controladores y ***se*** combinan los grupos moviéndose hacia arriba por la estructura del programa.

La integración sigue el esquema ilustrado en la Figura 18.7. Se combinan los módulos para formar los grupos 1, 2 y **3.** Cada uno de los grupos se somete a prueba

mediante un controlador (mostrado como un bloque punteado). Los módulos de los grupos 1 y 2 son subordinados de M_a. Los controladores D_1 y D_2 se eliminan y los grupos interaccionan directamente con M_a. De forma similar, se elimina el controlador D_3 del grupo **3** antes de la integración con el módulo M_b. Tanto M_a como **M_b** se integrarán finalmente con el módulo M_c y así sucesivamente.

PUNTO CLAVE

La integración ascendente elimina la necesidad de resguardos complejos.

A medida que la integración progresa hacia arriba, disminuye la necesidad de controladores de prueba diferentes. De hecho, si los dos niveles superiores de la estructura del programa se integran de forma descendente, se puede reducir sustancialmente el número de controladores y se simplifica enormemente la integración de grupos.

18.4.3. Prueba de regresión

Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software cambia. Se establecen nuevos caminos de flujo de datos, pueden ocurrir nuevas E/S y se invoca una nueva lógica de control. Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados.

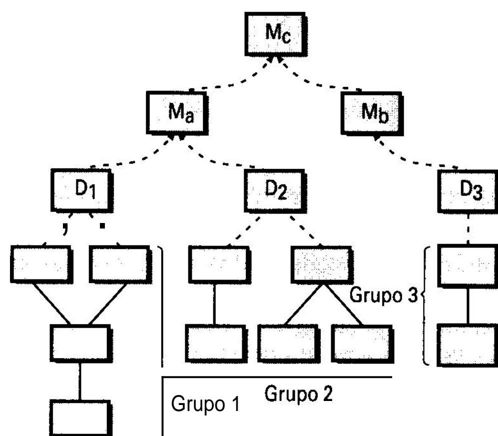


FIGURA 18.7. Integración ascendente.

En un contexto más amplio, las pruebas con éxito (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores hay que corregirlos. Cuando se corrige el software, se cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que lo soportan). La prueba de regresión es la actividad que ayuda a asegurar que los cambios (debidos a las pruebas o por otros motivos) no introducen un comportamiento no deseado o errores adicionales.

La prueba de regresión es una estrategia importante para reducir «efectos colaterales». Se deben ejecutar pruebas de regresión cada vez que se realice un cambio importante en el software (incluyendo la integración de nuevos módulos).

La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas de reproducción de captura. Las *herramientas de reproducción **de** captura* permiten al ingeniero del software capturar casos de prueba y los resultados para la subsiguiente reproducción y comparación. El conjunto de pruebas de regresión (el subconjunto de pruebas a realizar) contiene tres clases diferentes de casos de prueba:

- una muestra representativa de pruebas que ejercite todas las funciones del software;
- pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio;
- pruebas que se centran en los componentes del **software** que han cambiado.

A medida que progresa la prueba de integración, **el** número de pruebas de regresión puede crecer demasiado. Por tanto, el conjunto de pruebas de regresión debería diseñarse para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

18.4.4. Prueba de humo

La prueba de humo es un método de prueba de integración que es comúnmente utilizada cuando se ha desarrollado un producto software «empaquetado». **Es** diseñado como un mecanismo para proyectos **críticos** por tiempo, permitiendo que el equipo de software valore su proyecto sobre una base sólida. En esencia, la prueba de humo comprende las siguientes actividades:

1. Los componentes software que han sido traducidos a código se integran en una «construcción». Una construcción incluye ficheros de datos, librerías, módulos reutilizables y componentes de ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para descrubir errores que impiden a la construcción realizar su fun-

ción adecuadamente. El objetivo será descubrir errores «bloqueantes» que tengan la mayor probabilidad de impedir al proyecto de software el cumplimiento de su planificación.

3. Es habitual en la prueba de humo que la construcción se integre con otras construcciones y que se aplica una prueba de humo al producto completo (en su forma actual). La integración puede hacerse bien de forma descendente *(top-down)* o ascendente *(bottom-up).*

PUNTO CLAVE

La prueba de humo se caracteriza por una estrategia de integración continua. El software es reconfigurado (con la incorporación de nuevos componentes) y utilizado continuamente.

La frecuencia continua de la prueba completa del producto puede sorprender a algunos lectores. En cualquier caso, las frecuentes pruebas dan a gestores y profesionales una valoración realista de la evolución de las pruebas de integración. McConnell [MCO96] describe la prueba de humo de la siguiente forma:

> La prueba de humo ejercita el sistema entero de principio a fin. No ha de ser exhaustiva, pero será capaz de descubrir importantes problemas. La prueba de humo será suficiente si verificamos de forma completa la construcción y podemos asumir que es suficientemente estable para ser probado con más profundidad.

La prueba de humo facilita una serie de beneficios cuando se aplica sobre proyectos de ingeniería del software complejos y críticos por su duración:

- *Se minimizan los riesgos de integración.* Dado que las pruebas de humo son realizadas frecuentemente, incompatibilidades y otros errores bloqueantes son descubiertos rápidamente, por eso se reduce la posibilidad de impactos importantes en la planificación por errores sin descubrir.
- *Se perfecciona la calidad del producto final.* Dado que la prueba de humo es un método orientado a la construcción (integración), es probable que descubra errores funcionales, además de defectos de diseño a nivel de componente y de arquitectura. Si estos defectos se corrigen rápidamente, el resultado será un producto de gran calidad.

Cita:

La construcción diaria es el aliciente del proyecto. Si no hay aliciente, el proyecto se muere.

Jim McCarthy

- *Se simplifican el diagnóstico y la corrección de errores.* Al igual que todos los enfoques de prueba de integración, es probable que los errores sin descubrir durante la prueba de humo se asocien a «nuevos incrementos de software» - esto es, el software que se acaba de añadir a la construcción es una posible causa de un error que se acaba de descubrir—.
- *El progreso es fácil de observar.* Cada día que pasa, se integra más software y se demuestra que funciona. Esto mejora la moral del equipo y da una indicación a los gestores del progreso que se está realizando.

18.4.5. Comentarios sobre la prueba de integración

Ha habido muchos estudios (por ejemplo, [BEI84]) sobre las ventajas y desventajas de la prueba de integración ascendente frente a la descendente. En general, las ventajas de una estrategia tienden a convertirse en desventajas para la otra estrategia. La principal desventaja del enfoque descendente es la necesidad de resguardos y las dificultades de prueba que pueden estar asociados con ellos. Los problemas asociados con los resguardos pueden quedar compensados por la ventaja de poder probar de antemano las principales funciones de control. La principal desventaja de la integración ascendente es que «el programa como entidad no existe hasta que se ha añadido el último módulo» [MYE79]. Este inconveniente se resuelve con la mayor facilidad de diseño de casos de prueba y con la falta de resguardos.

¿Qué es un módulo crítico y por qué debemos identificarlo?

La selección de una estrategia de integración depende de las características del software y, a veces, de la planificación del proyecto. En general, el mejor compromiso puede ser un enfoque combinado (a veces denominado *prueba sandwich)* que use la descendente para los niveles superiores de la estructura del programa, junto con la ascendente para los niveles subordinados.

A medida que progresa la prueba de integración, el responsable de las pruebas debe identificar los módulos críticos. Un módulo crítico es aquel que tiene una o más de las siguientes características: (1) está dirigido a varios requisitos del software; (2) tiene un mayor nivel de control (está relativamente alto en la estructura del programa); (3) es complejo o propenso a errores (se puede usar la complejidad ciclomática como indicador); o (4) tiene unos requisitos de rendimiento muy definidos. Los módulos críticos deben probarse lo antes posible. Además, las pruebas de regresión se deben centrar en el funcionamiento de los módulos críticos.

18.5 PRUEBA DE VALIDACIÓN

Tras la culminación de la prueba de integración, el software está completamente ensamblado como un paquete, se han encontrado y corregido los errores de interfaz y puede comenzar una serie final de pruebas del software: la *prueba* de validación. La validación puede definirse de muchas formas, pero una simple (aunque vulgar) definición es que la validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente. En este punto, un desarrollador de software estricto podría protestar: «¿Qué o quién es el árbitro de las expectativas razonables?»

PUNTO CLAVE

Como en otras etapas de la prueba, la validación permite descubrir errores, pero su enfoque está en el nivel de requisitos —sobre cosas que son necesarias para el usuario final—.

Las expectativas razonables están definidas en la Especificación de Requisitos del Software —un documento (Capítulo 12) que describe todos los atributos del software visibles para el usuario. La especificación contiene una sección denominada—. «Criterios de validación». La información contenida en esa sección forma la base del enfoque a la prueba de validación.

18.5.1. Criterios de la prueba de validación

La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Un plan de prueba traza la clase de pruebas que se han de llevar a cabo, y un procedimiento de prueba define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos. Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que la documentación es correcta e inteligible y que se alcanzan otros requisitos (por ejemplo, portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento).

Una vez que se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes: (1) las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables; o (2) se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada. A menudo es necesario negociar con el cliente un método para resolver las deficiencias.

18.5.2. Revisión de la configuración

Un elemento importante del proceso de validación es la revisión de *la* configuración. La intención de la revisión es asegurarse de que todos los elementos de la configuración del software se han desarrollado apropiadamente, se han catalogado y están suficientemente detallados para soportar la fase de mantenimiento durante el ciclo de vida del software. La revisión de la configuración, a veces denominada *auditoría*, se ha estudiado con más detalle en el Capítulo 9.

18.5.3. Pruebas alfa y beta

Es virtualmente imposible que un desarrollador de software pueda prever cómo utilizará el usuario realmente el programa. Se pueden malinterpretar las instrucciones de uso, se pueden utilizar habitualmente extrañas combinaciones de datos, y una salida que puede parecer clara para el responsable de las pruebas y puede ser ininteligible para el usuario.

Cuando se construye software a medida para un cliente, se llevan a cabo una serie de pruebas de aceptación para permitir que el cliente valide todos los requisitos. Las realiza el usuario final en lugar del responsable del desarrollo del sistema, una prueba de aceptación puede ir desde un informal «paso de prueba» hasta la ejecución sistemática de una serie de pruebas bien planificadas. De hecho, la prueba de aceptación puede tener lugar a lo largo de semanas o meses, descubriendo así errores acumulados que pueden ir degradando el sistema.

Si el software se desarrolla como un producto que va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales para cada uno de ellos. La mayoría de los desarrolladores de productos de software llevan a cabo un proceso denominado prueba alfa y beta para descubrir errores que parezca que sólo el usuario final puede descubrir.

La prueba *alfa* se lleva a cabo, por un cliente, en el lugar de desarrollo. Se usa el software de forma natural con el desarrollador como observador del usuario y registrando los errores y los problemas de uso. Las pruebas alfa se llevan a cabo en un entorno controlado.

La *prueba beta* se lleva a cabo por los usuarios finales del software en los lugares de trabajo de los clientes. A diferencia de la prueba alfa, el desarrollador no está presente normalmente. Así, la prueba beta es una aplicación «en vivo» del software en un entorno que no puede ser controlado por el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que encuentra durante la prueba beta e informa a intervalos regulares al desarrollador. Como resultado de los problemas informados durante la prueba beta, el desarrollador del software lleva a cabo modificaciones y así prepara una versión del producto de software para toda la clase de clientes.

18.6 PRUEBA DEL SISTEMA

Al comienzo de este libro, pusimos énfasis en el hecho de que el software es sólo un elemento de un sistema mayor basado en computadora. Finalmente, el software es incorporado a otros elementos del sistema (por ejemplo, nuevo hardware, información) y realizan una serie de pruebas de integración del sistema y de validación. Estas pruebas caen fuera del ámbito del proceso de ingeniería del software y no las realiza únicamente el desarrollador del software. Sin embargo, los pasos dados durante el diseño del software y durante la prueba pueden mejorar enormemente la probabilidad de éxito en la integración del software en el sistema.

Cita:

Semejante a la muerte y a los impuestos, la prueba es desagradable e inevitable.

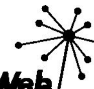
Ed Yourdon

Un problema típico de la prueba del sistema es la ((delegaciónde culpabilidad». Esto ocurre cuando se descubre un error y cada uno de los creadores de cada elemento del sistema echa la culpa del problema a los otros. En vez de verse envuelto en esta absurda situación, el ingeniero del software debe anticiparse a los posibles problemas de interacción y: (1) diseñar caminos de manejo de errores que prueben toda la información procedente de otros elementos del sistema; (2) llevar a cabo una serie de pruebas que simulen la presencia de datos en mal estado o de otros posibles errores en la interfaz del software; (3) registrar los resultados de las pruebas como «evidencia» en el caso de que se le señale con el dedo; (4) participar en la planificación y el diseño de pruebas del sistema para asegurarse de que el software se prueba de forma adecuada.

La prueba del sistema, realmente, está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. En las siguientes secciones examinamos los tipos de pruebas del sistema [BEI84] valiosos para sistemas basados en software.

18.6.1. Prueba de recuperación

Muchos sistemas basados en computadora deben recuperarse de los fallos y continuar el proceso en un tiempo previamente especificado. En algunos casos, un sistema debe ser tolerante con los fallos; es decir, los fallos del proceso no deben hacer que cese el funcionamiento de todo el sistema.



Referencia Web

Amplia información sobre la prueba del software y su relación con las necesidades de calidad pueden obtenerse en **www.stqe.net**

En otros casos, se debe corregir un fallo del sistema en un determinado periodo de tiempo para que no se produzca un serio daño económico.

La *prueba de recuperación* es una prueba del sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo apropiadamente. Si la recuperación es automática (llevada a cabo por el propio sistema) hay'que evaluar la corrección de la inicialización, de los mecanismos de recuperación del estado del sistema, de la recuperación de datos y del proceso de rearranque. Si la recuperación requiere la intervención humana, hay que evaluar los tiempos medios de reparación (TMR) para determinar si están dentro de unos límites aceptables.

18.6.2. Prueba de seguridad

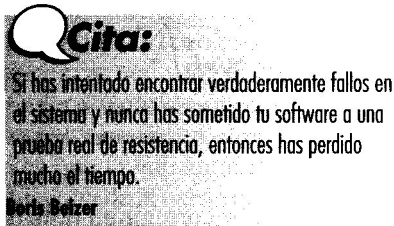
Cualquier sistema basado en computadora que maneje información sensible o lleve a cabo acciones que puedan perjudicar (o beneficiar) impropiamente a las personas es un posible objetivo para entradas impropias o ilegales al sistema. Este acceso al sistema incluye un amplio rango de actividades: *«piratas informáticos»* que intentan entrar en los sistemas por deporte, empleados disgustados que intentan penetrar por venganza e individuos deshonestos que intentan penetrar para obtener ganancias personales ilícitas.

La *prueba de seguridad* intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de accesos impropios. Para citar a Beizer [BEI84]: «Por supuesto, la seguridad del sistema debe ser probada en su invulnerabilidad frente a un ataque frontal, pero también debe probarse en su invulnerabilidad a ataques por los flancos o por la retaguardia.»

Durante la prueba de seguridad, el responsable de la prueba desempeña el papel de un individuo que desea entrar en el sistema. ¡Todo vale! Debe intentar conseguir las claves de acceso por cualquier medio, puede atacar al sistema con software a medida, diseñado para romper cualquier defensa que se haya construido, debe bloquear el sistema, negando así el servicio a otras personas, debe producir a propósito errores del sistema, intentando acceder durante la recuperación o debe curiosear en los datos sin protección, intentando encontrar la clave de acceso al sistema, etc.

Con tiempo y recursos suficientes, una buena prueba de seguridad terminará por acceder al sistema. **El**

papel del diseñador del sistema es hacer que el coste de la entrada ilegal sea mayor que el valor de la información obtenida.



18.6.3. Prueba de resistencia (Stress)

Durante los pasos de prueba anteriores, las técnicas de caja blanca y de caja negra daban como resultado la evaluación del funcionamiento **y** del rendimiento normales del programa. Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales. En esencia, el sujeto que realiza la prueba de resistencia se pregunta: «¿A qué potencia puedo ponerlo a funcionar antes de que falle?»

La *prueba de resistencia* ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Por ejemplo: (1) diseñar pruebas especiales que generen diez interrupciones por segundo, cuando las normales son una o dos; (2) incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada; *(3)* ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos; **(4)** diseñar casos de prueba que puedan dar problemas en un sistema operativo virtual o *(5)* diseñar casos de prueba que produzcan excesivas búsquedas de datos residentes en disco. Esencialmente, el responsable de la prueba intenta romper el programa.

Una variante de la prueba de resistencia es una técnica denominada prueba de *sensibilidad.* En algunas situaciones (la más común se da con algoritmos matemáticos), un rango de datos muy pequeño dentro de los límites de una entrada válida para un programa puede producir un proceso exagerado e incluso erróneo o una profunda degradación del rendimiento. Esta situación es análoga a una singularidad en una función matemática. La prueba de sensibilidad intenta descubrir combinaciones de datos dentro de una clase de entrada válida que pueda producir inestabilidad o un proceso incorrecto.

18.6.4. Prueba de rendimiento

Para sistemas de tiempo real y sistemas empotrados, es inaceptable el software que proporciona las funciones requeridas pero no se ajusta a los requisitos de rendimiento. La *prueba de rendimiento* está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. La prueba de rendimiento se da durante todos los pasos del proceso de la prueba. Incluso al nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de caja blanca. Sin embargo, hasta que no están completamente integrados todos los elementos del sistema no se puede asegurar realmente el rendimiento del sistema.

Las pruebas de rendimiento, a menudo, van emparejadas con las pruebas de resistencia y, frecuentemente, requieren instrumentación tanto de software como de hardware. Es decir, muchas veces es necesario medir la utilización de recursos (por ejemplo, ciclos de procesador), de un modo exacto. La instrumentación externa puede monitorizar los intervalos de ejecución, los sucesos ocurridos (por ejemplo, interrupciones) y muestras de los estados de la máquina en un funcionamiento normal. Instrumentando un sistema, el encargado de la prueba puede descubrir situaciones que lleven a degradaciones y posibles fallos del sistema.

18.7 EL ARTE DE LA DEPURACIÓN

La prueba del software es un proceso que puede planificarse y especificarse sistemáticamente. Se puede llevar a cabo el diseño de casos de prueba, se puede definir una estrategia y se pueden evaluar los resultados en comparación con las expectativas prescritas.

La *depuración* ocurre como consecuencia de una prueba efectiva. Es decir, cuando un caso de prueba descubre un error, la depuración es el proceso que provoca la eliminación del error. Aunque la depuración puede y debe ser un proceso ordenado, sigue teniendo mucho de arte. Un ingeniero del software, al evaluar los resultados de una prueba, se encuentra frecuentemente con una indicación «sintomática» de un problema en el software. Es decir, la manifestación externa de un error, y la causa interna del error pueden no estar relacionados de una forma obvia. El proceso mental, apenas comprendido, que conecta un síntoma con una causa es la depuración.

BugNet facilita información sobre problemas de seguridad y fallos en software basado en PC y proporciona una información útil sobre temas de depuración: **www.bugnet.com**

18.7.1. El Proceso de depuración

La depuración no es una prueba, pero siempre ocurre como consecuencia de la prueba[4]. Como se muestra en la Figura 18.8, el proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados y aparece una falta de correspondencia entre los esperados y los encontrados realmente. En muchos casos, los datos que no concuerdan son un síntoma de una causa subyacente que todavía permanece oculta. El proceso de depuración intenta hacer corresponder el sistema con una causa, llevando así a la corrección del error.

El proceso de depuración siempre tiene uno de los dos resultados siguientes: (1) se encuentra la causa, se corrige y se elimina; o (2) no se encuentra la causa. En este último caso, la persona que realiza la depuración debe sospechar la causa, diseñar un caso de prueba que ayude a confirmar sus sospechas y el trabajo vuelve hacia atrás a la corrección del error de una forma iterativa.

¿Por qué es tan difícil la depuración? Todo parece indicar que la respuesta tiene más que ver con la psicología humana (véase la siguiente sección) que con la tecnología del software. Sin embargo, varias características de los errores nos dan algunas pistas:

1. El síntoma y la causa pueden ser geográficamente remotos entre sí. Es decir, el síntoma puede aparecer en una parte del programa, mientras que la causa está localizada en otra parte muy alejada. Las estructuras de programa fuertemente acopladas (Capítulo 13) resaltan esta situación.
2. El síntoma puede desaparecer (temporalmente) al corregir otro error.
3. El síntoma puede realmente estar producido por algo que no es un error (por ejemplo, inexactitud en los redondeos).
4. El síntoma puede estar causado por un error humano que no sea fácilmente detectado.
5. El síntoma puede ser el resultado de problemas de temporización en vez de problemas de proceso.
6. Puede ser difícil reproducir exactamente las condiciones de entrada (por ejemplo, una aplicación de tiempo real en la que el orden de la entrada no está determinado).
7. El síntoma puede aparecer de forma intermitente. Esto es particularmente comente en sistemas empotrados que acoplan el hardware y el software de manera confusa.
8. El síntoma puede ser debido a causas que se distribuyen por una serie de tareas ejecutándose en diferentes procesadores [CHE90].

Cita:

La variedad que dentro de los programas de computador debe diagnosticarse [para depuración] es probablemente mayor que la variedad en otros ejemplos de sistemas que son regularmente diagnosticados.

John Gould

Durante la depuración encontramos errores que van desde lo ligeramente inesperado (por ejemplo, un formato de salida incorrecto) hasta lo catastrófico (por ejemplo, el sistema falla, produciéndose serios daños económicos o físicos). A medida que las consecuencias de un error aumentan, crece la presión por encontrar su causa. A menudo la presión fuerza a un ingeniero del software a corregir un error introduciendo dos más.

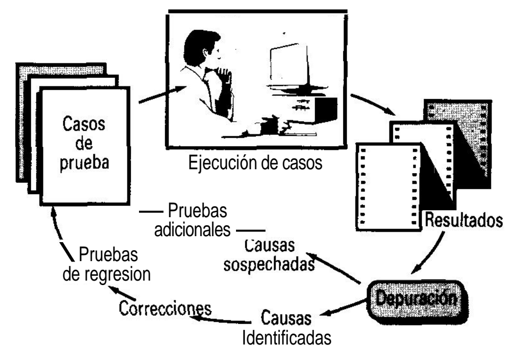


FIGURA 18.8. El proceso de depuración.

18.7.2. Consideraciones psicológicas

Desafortunadamente, todo parece indicar que la habilidad en la depuración es un rasgo innato del ser humano. A ciertas personas se les da bien y a otras no. Aunque las manifestaciones experimentales de la depuración están abiertas a muchas interpretaciones, se han detectado grandes variaciones en la destreza para la depuración de distintos programadores con el mismo bagaje de formación y de experiencia.

Hablando de los aspectos humanos de la depuración, Shneiderman [SHN80] manifiesta:

> La depuración es una de las partes más frustrantes de la programación. Contiene elementos de resolución de problemas o de rompecabezas, junto con el desagradable reconocimiento de que se ha cometido un error. La enorme ansiedad y la **no** inclinación a aceptar la posibilidad de cometer errores hace que la tarea sea extremadamente difícil. Afortunadamente, también se da un gran alivio y disminuye la tensión cuando el error **es** finalmente...corregido.

Aunque puede resultar difícil «aprender» a depurar, se pueden proponer varios enfoques del problema. En la siguiente sección los examinamos.

[4] Al decir esta frase, tomamos el punto de vista más amplio que se puede tener de la prueba. No sólo ha de llevar a cabo la prueba el equipo de desarrollo antes de entregar el software, sino que el cliente/usuario prueba el software cada vez que lo usa.

18.7.3. Enfoques de la depuración

Independientemente del enfoque que se utilice, la depuración tiene un objetivo primordial: encontrar y corregir la causa de un error en el software. El objetivo se consigue mediante una combinación de una evaluación sistemática, de intuición y de suerte. Bradley [BRA85] describe el enfoque de la depuración de la siguiente forma:

> La depuración es una aplicación directa del método científico desarrollado hace 2.500 años. La base de la depuración es la localización de la fuente del problema [la causa] mediante partición binaria, manejando hipótesis que predigan nuevos valores a examinar.
>
> Tomemos un sencillo ejemplo que no tiene que ver con el software: en mi casa no funciona una lámpara. Si no funciona nada en la casa, la causa debe estar en el circuito principal de fusibles o fuera de la casa; miro fuera para ver si hay un apagón en todo el vecindario. Conecto la sospechosa lámpara a un enchufe que funcione y un aparato que funcione en el circuito sospechoso. **Así** se sigue la secuencia de hipótesis y de pruebas.

En general, existen tres enfoques que se pueden proponer para la depuración [MYE79]:

1. Fuerza bruta
2. Vuelta atrás
3. Eliminación de causas

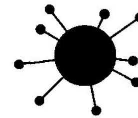
La categoría de depuración por la *fuerza bruta* es probablemente la más común y menos eficiente a la hora de aislar la causa del error en el software. Aplicamos los métodos de depuración por fuerza bruta cuando todo lo demás falla. Mediante una filosofía de «dejar que la computadora encuentre el error», se hacen volcados de memoria, trazas de ejecución y se cargan multitud de sentencias *Mostrar* en el programa. Esperamos que en algún lugar de la gran cantidad de información generada encontremos alguna pista que nos lleve a la causa de un error. Aunque la gran cantidad de información producida nos puede llevar finalmente al éxito, lo más frecuente es que se desperdicie tiempo y esfuerzo. ¡Primero se debe usar la inteligencia!

fíjate un tiempo limitado, por ejemplo dos horas, en relación a la cantidad de tiempo que tu inviertes para depurar un problema de tu incumbencia. Después qué, ¡pide ayuda!

La *vuelta atrás* es un enfoque más normal para la depuración, que se puede usar con éxito para pequeños programas. Partiendo del lugar donde se descubre el síntoma, *se* recorre hacia atrás (manualmente) el código fuente hasta que se llega a la posición de error. Desgraciadamente, a medida que aumenta el número de líneas del código, el número de posibles caminos de vuelta se hace difícilmente manejable.

El tercer enfoque para la depuración —*eliminación de causas*— se manifiesta mediante inducción o deducción e introduce el concepto de partición binaria. Los datos relacionados con la ocurrencia del error se organizan para aislar las posibles causas. Se llega a una «hipótesis de causa» y se usan los datos anteriores para probar o revocar la hipótesis. Alternativamente, se desarrolla una lista de todas las posibles causas y se llevan a cabo pruebas para eliminar cada una. Si alguna prueba inicial indica que determinada hipótesis de causa en particular parece prometedora, se refinan los datos con el fin de intentar aislar el error.

Cada uno de los enfoques anteriores puede complementarse con herramientas de depuración. Podemos usar una gran cantidad de compiladores de depuración, ayudas dinámicas para la depuración («trazadores»), generadores automáticos de casos de prueba, volcados de memoria y mapas de referencias cruzadas. Sin embargo, las herramientas no son un sustituto de la evaluación cuidadosa basada en un completo documento del diseño del software y un código fuente claro.



Herramientas CASE
Pruebo y Depuración.

Cualquier discusión sobre los enfoques para la depuración y sus herramientas no estaría completa sin mencionar un poderoso aliado: ¡otras personas! Cualquiera de nosotros podrá recordar haber estado dando vueltas en la cabeza durante horas o días a un error persistente. Desesperados, le explicamos el problema a un colega con el que damos por casualidad y le mostramos el listado. Instantáneamente (parece), se descubre la causa del error. Nuestro colega se aleja sonriendo ladinamente. Un punto de vista fresco, no embotado por horas de frustración, puede hacer maravillas. Una máxima final para la depuración puede ser: «¡Cuando todo lo demás falle, pide ayuda!»

Una vez que se ha encontrado un error, hay que corregirlo. Pero como ya hemos podido observar, la corrección de un error puede introducir otros errores y hacer más mal que bien.

? Cuando corrijo un error, ¿qué cuestiones debo preguntarme a mi mismo?

Van Vleck [VAN89] sugiere tres preguntas sencillas que debería preguntarse todo ingeniero del software antes de hacer la «corrección» que elimine la causa del error:

1 *¿Se repite la causa del error en otra parte del programa?* En muchas situaciones, el defecto de un programa está producido por un patrón de lógica erróneo que se puede repetir en cualquier lugar. La consideración explícita del patrón lógico puede terminar en el descubrimiento de otros errores.

2. *¿Cuál es el «siguiente error» que se podrá presentar a raíz de la corrección que hoy voy a realizar?* Antes de hacer la corrección, se debe evaluar el código fuente (o mejor, el diseño) para determinar el emparejamiento de la lógica y las estructuras de datos. Si la corrección se realiza en una sección del programa altamente acoplada, se debe tener cuidado al realizar cualquier cambio.

3. *¿Qué podríamos haber hecho para prevenir este error la primera vez?* Esta pregunta es el primer paso para establecer un método estadístico de garantía de calidad del software (Capítulo **8**). Si corregimos tanto el proceso como el producto, se eliminará el error del programa actual y se puede eliminar de todos los futuros programas.

RESUMEN

La prueba del software contabiliza el mayor porcentaje del esfuerzo técnico del proceso de desarrollo de software. Todavía estamos comenzando a comprender las sutilezas de la planificación sistemática de la prueba, de su ejecución y de su control.

El objetivo de la prueba de software es descubrir errores. Para conseguir este objetivo, se planifica y se ejecutan una serie de pasos; pruebas de unidad, de integración, de validación y del sistema. Las pruebas de unidad y de integración se centran en la verificación funcional de cada módulo y en la incorporación de los módulos en una estructura de programa. La prueba de validación demuestra el seguimiento de los requisitos del software y la prueba del sistema valida el software una vez que se ha incorporado en un sistema superior.

Cada paso de prueba se lleva a cabo mediante una serie de técnicas sistemáticas de prueba que ayudan en el diseño de casos de prueba. Con cada paso de prueba se amplía el nivel de abstracción con el que se considera el software.

A diferencia de la prueba (una actividad sistemática y planificada), la depuración se puede considerar un arte. **A** partir de una indicación sintomática de un problema, la actividad de la depuración debe rastrear la causa del error. De entre los recursos disponibles durante la depuración, el más valioso puede ser el apoyo de otros ingenieros de software.

El requisito de que el software sea cada vez de mayor calidad exige un planteamiento más sistemático de la prueba. Citando a Dunn y Ullman [DUN82]:

> Lo que se requiere es una estrategia global, que se extienda por el espacio estratégico de la prueba, tan deliberada en su metodología como lo fue el desarrollo sistemático en el que se basaron el análisis, el diseño y la codificación.

En este capítulo hemos examinado el espacio estratégico de la prueba, considerando los pasos que tienen la mayor probabilidad de conseguir el fin Último de la prueba: encontrar y subsanar los defectos de una manera ordenada **y** efectiva.

REFERENCIAS

[BEI84] Beizer, B., *Software System Testing and Quality Assurance,* Van Nostrand Reinhold, 1984.

[BOE81] Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981, p. 37.

[BRA85] Bradley, J.H., «The science and Art of Debugging», Computerworld, 19 de Agosto de 1985, pp. 35-38.

[CHE90] Cheung, W. H., J. P. Black y E. Manning, «A Framework for Distributed Debugging», *IEEE Software*, Enero 1990, pp. 106-115.

[DUN82] Dunn, R., y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982, p. 158.

[GIL95] Gilb, T., «What We Fail To Do In Our Current Testing Culture», *Testing Techniques Newsletter,* (edición en-línea, ttn@soft.com), Software Research, Inc, San Francisco, Enero 199.5.

[MCO96] McConnell, S., «Best Practices: Daily Build and Smoke Test», IEEE Software, vol. 13, n.º 4, Julio 1996, pp. 143-144.

[MILL77] Miller, E., «The Philosophy of Testing», *Program Testing Techniques,* IEEE Computer Society Press, 1977, pp. 1-3.

[MUS89] Musa, J. D., y Ackerman, A. F., «Quantifying Software Validation: When to Stop Testing?», *IEEE Software,* Mayo 1989, pp. 19-27.

[MYE79] Myers, G., *The Art of Software Tests,* Wiley, 1979.

[SHO83] Shooman, **M.** L., *Software Engineering,* Mcgraw-Hill, 1983.

[SHN80] Shneiderman, B., *Software Psychology,* Winthrop Publishers, 1980, p. 28.

[VAN89] Van Bleck, T., «Three Questions About Each Bug You find», *ACM Sofiare Engineering Notes,* vol. 14, n.º 5, Julio 1989, pp. 62-63.

[WAL89] Wallance, D. R., y R. U. Fujii, «Software Verification and Validation: An Overview», *IEEE Software*, Mayo 1989, pp. 10-17.

[YOU75] Yourdon, E., *Techniques of Progrum Structure and Design*, Prentice-Hall, 1975.

PROBLEMAS Y PUNTOS A CONSIDERAR

18.1. Con sus propias palabras, describa las diferencias entre verificación y validación. ¿Utilizan las dos los métodos de diseño de casos de prueba y las estrategias de prueba?

18.2. Haga una lista de algunos problemas que puedan estar asociados con la creación de un grupo independiente de prueba. ¿Están formados por las mismas personas el GIP y el grupo SQA?

18.3. ¿Es siempre posible desarrollar una estrategia de prueba de software que use la secuencia de pasos de prueba descrita en la Sección 18.1.3? ¿Qué posibles complicaciones pueden surgir para sistemas empotrados?

18.4. Si sólo pudiera seleccionar tres métodos de diseño de casos de prueba para aplicarlos durante la prueba de unidad, ¿cuáles serían y por qué?

18.5. Porqué es difícil de realizar la prueba unitaria a un módulo con alto nivel de integración.

18.6. Desarrolle una estrategia de prueba de integración para cualquiera de los sistemas implementados en los problemas 16.4 a 16.11. Defina las fases de prueba, indique el orden de integración, especifique el software de prueba adicional y justifique el orden de integración. Suponga que todos los módulos han sido probados en unidad y están disponibles. [Nota: puede ser necesario comenzar trabajando un poco con el diseño inicialmente.]

18.7. ¿Cómo puede afectar la planificación del proyecto a la prueba de integración?

18.8. ¿Es posible o incluso deseable la prueba de unidad en cualquier circunstancia? Ponga ejemplos que justifiquen su respuesta.

18.9. ¿Quién debe llevar a cabo la prueba de validación —el desarrollador del software o el usuario—? Justifique su respuesta.

18.10. Desarrolle una estrategia de prueba completa para el sistema *HogarSeguro* descrito anteriormente en este libro. Documéntela en una *Especificación de Prueba.*

18.11. Como proyecto de clase, desarrolle una guía de depuración para su instalación. La guía debe proporcionar consejos orientados al lenguaje y al sistema que se hayan aprendido con las malas experiencias. Comience con un esbozo de los temas que se tengan que revisar por la clase y **su** profesor. Publique la guía para otras personas de su entorno.

OTRAS LECTURAS Y FUENTES DE INFORMACIÓN

Libros orientados a las estrategias de prueba del software son los de Black (*Managing the Testing Process,* Microsoft Press, 1999), Dustin, Rashka and Paul (*Test Process Improvement:* Step-By-Step Guide to Structured Testing, Addison-Wesley, 1999), Perry (*Suwiving the Top Ten Challenges of Software Testing: A People-Oriented Approach,* Dorset House, 1997), and Kit and Finzi (*Software Testing in the Real World: Improving the Process,* Addison-Wesley, 1995).

Kaner, Nguyen y Falk (*Testing Computer Softwa-re,* Wiley, 1999), Hutcheson (*Software Testing Methods and Metrics: The Most Important Tests,* McGraw Hill, 1997), Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing,* Prentice Hall, 1995), Jorgensen (*Software Testing: A Crafts-man 's Approach,* CRC Press, 1995) presentan estudios sobre los métodos y estrategias de prueba.

Además, antiguos libros de Evans (*Productive Software Test Management*, Wiley-Interscience, 1984), Hetzel (*The Complete Guide to Software Testing,* QED Information Sciences, 1984), Beizer [BEI84], Ould y Unwin (*Testing in Software Development,* Cambridge University Press, 1986), Marks (*Testing Very Big Systems*, McGraw-Hill, 1992), y Kaner et al. (*Testing Computer Software,* 2.ª ed., Van Nostrand Reinhold, 1993) define los pasos para una estrategia efectiva, proporciona un conjunto de técnicas y directrices y sugiere procedimientos para controlar y hacer un seguimiento a los procesos de prueba. Hutcheson (*Software Testing Methods and Metrics,* McGraw-Hill, 1996) presenta unos métodos y estrategias de prueba pero también proporciona un estudio detallado de cómo se puede usar la medición para conseguir una prueba efectiva.

Un libro de Dunn (*Software Defect Removal,* McGraw-Hill, 1984) contiene unas directrices para la depuración. Beizer [BEI84] presenta una interesante «taxonomía de errores» que puede conducir a unos métodos efectivos para la planificación de pruebas. McConnell (*Code Complete,* Microsoft Press, 1993) presenta unos pragmáticos consejos sobre las pruebas de unidad y de integración así como sobre la depuración.

Una amplia variedad de fuentes de información sobre pruebas del software y elementos relacionados están disponibles en Internet. Una lista actualizada de páginas web sobre conceptos de prueba, métodos y estrategias pueden encontrarse en **http://www.pressman5.com.**