UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Professional Testing Master

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

UTN.BA UNIVERSIDAD TECNOLÓGICA NACIONAL FACULTAD REGIONAL BUENOS AIRES | Centro de e-Learning

Unidad 1: Fundamentos de testing

Presentación:

En esta Primera Unidad del curso, explicaremos los conceptos y problemas fundamentales de software testing, así como también la importancia de cada una de las fases de la actividad usando un enfoque estratégico.

Objetivos:

Al terminar la Unidad los participantes:

Se apropiarán de los principales conceptos relacionados con software testing y comprenderán los desafíos que impone la actividad.

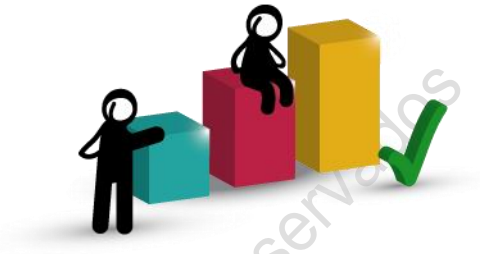
Habrán conocido y asimilado los principios y objetivos que guían la práctica de software testing.

Conocerán las principales prácticas de testing llevadas a cabo en todo proyecto de software.

Bloques temáticos:

1. Introducción a software testing
2. Conceptos fundamentales
3. Importancia del testing
4. Aspectos psicológicos
5. Aspectos económicos
6. Objetivos y limitaciones del testing
7. Un enfoque estratégico

Contenido



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

**** El MEC es el modelo de E-learning colaborativo de nuestro Centro.***

Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1 Introducción

La capacidad de la etapa de testing para detectar errores en forma temprana, influye en gran medida en el éxito de un proyecto de software. Esto da una razón más que importante para focalizarnos en los aspectos fundamentales de la actividad.

Antes que todo, veamos los objetivos principales que abordaremos, lo cuales son que los participantes

- Comprendan la importancia del testing para reducir el riesgo de un proyecto de software.
- Se formen en los conceptos fundamentales del software testing
- Aprendan a aplicar un enfoque estratégico para el testing de un proyecto de software

1.1 Contexto

Durante las últimas tres décadas, el testing ha representado más del 50% del esfuerzo y el costo de los proyectos de desarrollo de software [Mye04]. No obstante, muchas veces es menospreciado como un elemento secundario en el proceso pese que a nivel mundial las organizaciones que emplean y desarrollan software reciben un gran impacto (desde pérdidas económicas hasta pérdida de imagen) por no realizar un correcto testing de sus aplicaciones de manera adecuada.

Por otro lado, en el ámbito académico, no es habitual que las Universidades u otras organizaciones educativas dicten materias dedicadas exclusivamente a software testing, con lo cual los profesionales de Sistemas egresan sin tener un conocimiento suficientemente profundo sobre este aspecto tan importante. Tampoco es habitual que los docentes aconsejen a los alumnos que recién se inician sobre cómo realizar el testing de sus propios ejercicios.

Por estos motivos consideramos apropiado el diseño de un curso para adquirir los fundamentos de software testing y formar a los participantes en las mejores prácticas de la actividad.

2 Conceptos fundamentales de testing

Existen numerosas historias espectaculares sobre las fallas de software en la última década. Incluso con estas lecciones evidentes sobre las consecuencias de software deficiente, las fallas de software siguen ocurriendo. Estos fallos costaron a la economía de EE.UU. un estimado de 60 mil millones de dólares por año [Eve07]. Aproximadamente 22 mil millones de las pérdidas anuales podrían haber sido eliminados mediante pruebas realizadas apropiadamente durante todas las fases de desarrollo de software.

Las siguientes secciones intentarán constituirse en una guía para lector, que mediante la exposición de las principales situaciones, problemas y soluciones que surgen en el contexto de software testing, aporte al menos un granito de arena que ayude a revertir esta situación.

2.1 ¿Qué es el testing de software?

Las pruebas de software (en inglés software testing) son las investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad de un producto de software. Intentan principalmente encontrar defectos en el producto. Estas pruebas pueden ser de dos tipos:

2.1.1 Pruebas estáticas

Son el tipo de pruebas que se realizan sin ejecutar el código de la aplicación. Es decir que no se ejecuta el programa o alguno de sus componentes.

Estas pruebas pueden referirse, por ejemplo, a la revisión de documentos (de diseño, requerimientos, entre otros) o bien del propio programa o alguno de sus componentes pero sin ejecutarlo. Esto se debe a que se pueden realizar "pruebas de escritorio" con el objetivo de seguir los flujos de la aplicación, o inspeccionar el código fuente en forma manual o automatizada para detectar inconvenientes, como malas prácticas, código potencialmente falible, incumplimiento de estándares, etc.

2.1.2 Pruebas dinámicas

Todas aquellas pruebas que para su ejecución requieren la ejecución de la aplicación.

Debido a la naturaleza dinámica de la ejecución de pruebas es posible medir con mayor precisión el comportamiento de la aplicación desarrollada. Por eso, este es el tipo de testing más común y en el cual se centrará mayormente este curso. Se basa fundamentalmente en concepto de caso prueba.

2.1.3 Caso de prueba

Definición (según el estándar IEEE 610): Conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y postcondiciones de ejecución, desarrollado con un objetivo en particular o condición de prueba, tales como probar un determinado camino de ejecución o para verificar el cumplimiento de un requisito determinado.

Ejemplo de caso de prueba para verificar la funcionalidad de borrado de un cliente en una aplicación de facturación:

Caso Nro	Título	Precondición	Entrada	Resultado esperado	Postcondición	Resultado obtenido
23	Eliminación de un cliente existente	El cliente 8179 existe en la base de datos	se ingresa el código 8179	Mensaje “El cliente fue eliminado”	El cliente 8179 ya no está en la base de datos	**OK**

Podemos decir entonces que el testing (actividad de prueba) de un sistema o software consiste en realizar la ejecución de uno o varios tests (pruebas), cada uno de los cuales consiste en la ejecución de uno o más casos de prueba y la verificación de los resultados.

2.2 Importancia del testing

Para demostrar la importancia del testing, se propone al estudiante la siguiente actividad. Escriba un conjunto de casos de prueba para testear un programa simple, es decir un conjunto de datos que el programa debería manejar correctamente para ser considerado exitoso. Ésta es la descripción del programa:

El programa lee tres valores enteros de un diálogo de entrada. Los tres valores representan las longitudes de los lados de un triángulo. El programa muestra un mensaje que indica si el triángulo es escaleno, isósceles o equilátero.

Recuerde que un triángulo escaleno es uno donde no hay dos lados que sean iguales, mientras que un triángulo isósceles tiene dos lados iguales, y un triángulo equilátero tiene tres lados de igual longitud. Por otra parte, todo triángulo debe cumplir que la suma de dos lados cualesquiera debe ser mayor que el lado restante.

Evalúe el conjunto de casos de prueba propuesto por usted, usándolo para responder a las siguientes preguntas. Dese un punto por cada "sí" como respuesta.

1. ¿Tiene un caso de prueba que representa un triángulo escaleno válido? (Tenga en cuenta que los casos de prueba tales como 1,2,3 y 2,5,10 no justifican una respuesta "sí", porque no existe un triángulo que tenga estas dimensiones)
2. ¿Tiene un caso de prueba que representa un triángulo equilátero válido?
3. ¿Tiene un caso de prueba que representa un triángulo isósceles válido? (Tenga en cuenta que un caso de prueba que represente 2,2,4 no contaría porque no es un triángulo válido.)
4. ¿Tiene por lo menos tres casos de prueba que representan triángulos isósceles válidos tales que incluyen las tres permutaciones de dos lados iguales (como, 3,3,4; 3,4,3 y 4,3,3)?
5. ¿Tiene un caso de prueba en el que un lado tiene un valor de cero?
6. ¿Tiene un caso de prueba en el que un lado tiene un valor negativo?
7. ¿Tiene un caso de prueba con tres números enteros mayores que cero tal que la suma de dos de los números es igual a la tercera? (Es decir, si el programa 1,2,3 dijo que representa un triángulo escaleno, sería un error.)
8. ¿Tiene por lo menos tres casos de prueba en la categoría 7 de tal manera que ha intentado las tres permutaciones posibles donde la longitud de un lado es igual a la suma de las longitudes de los otros dos lados (por ejemplo, 1,2,3; 1,3,2, y 3,1,2)?
9. ¿Tiene un caso de prueba con tres números enteros mayores que cero tal que la suma de dos de los números es menor que la tercera (tales como 1,2,4 o 12,15,30)?

10. ¿Tiene por lo menos tres casos de prueba en la categoría 9 de modo que ha intentado las tres permutaciones (por ejemplo, 1,2,4; 1,4,2 y 4,1,2)?
11. ¿Tiene un caso de prueba en el que todos los lados son iguales a cero (0,0,0)?
12. ¿Tiene por lo menos un caso de prueba especificando valores no enteros (como 2.5,3.5,5.5)?
13. ¿Tiene por lo menos un caso de prueba con el número incorrecto de valores (dos en lugar de tres números enteros, por ejemplo)?
14. ¿Para cada caso de prueba especificó el resultado esperado del programa, además de los valores de entrada?

Por supuesto, un conjunto de casos de prueba que satisfaga estas condiciones no garantiza que se pueden encontrar todos los errores posibles, pero ya que las preguntas 1 a 13 representan los errores que de hecho se han producido en diferentes versiones de este programa, una prueba adecuada de este programa debe exponer al menos estos errores.

Ahora, antes de preocuparse por su puntaje, considere lo siguiente: programadores profesionales altamente calificados puntuaron, en promedio, sólo 7,8 de un máximo de 14. Si usted lo hizo mejor, felicitaciones, sino, este curso intentará ayudarlo.

El objetivo del ejercicio es ilustrar que la prueba de incluso un programa trivial como este no es una tarea fácil. Y si esto es así, imagine la dificultad de poner a prueba un sistema de control de tráfico aéreo con 100.000 instrucciones, un compilador, o incluso un programa de sueldos estándar. Las pruebas también se hacen más difíciles con los lenguajes orientados a objetos, como Java y C++. Por ejemplo, los casos de prueba para aplicaciones hechas con estos lenguajes deben exponer los errores asociados con instancias de objetos y gestión de memoria.

2.3 Aspectos psicológicos del testing

Una de las causas principales de pruebas (testing) de software deficientes es el hecho de que la mayoría de los programadores comienzan con una definición falsa de la palabra. Ellos suelen decir:

- "La prueba es el procedimiento para demostrar que no hay errores en el programa."

O sino

- "El propósito de la prueba es demostrar que un programa realiza las funciones previstas correctamente."

O también

- "La prueba es el proceso de establecer confianza en que un programa hace lo que se supone que debe hacer."

Estas definiciones están al revés.

Cuando se prueba un programa se desea agregar algún valor al mismo. Agregar valor a través del testing significa elevar la calidad y confiabilidad del programa. Elevar la confiabilidad del programa quiere decir encontrar y eliminar errores.

Por lo tanto, usted no debe probar un programa para demostrar que funciona, sino que debe comenzar con la suposición de que el programa contiene errores (una suposición válida para casi cualquier programa) y luego probar el programa para encontrar la mayor cantidad posible de errores.

Por lo tanto, una definición más apropiada es la siguiente:

La prueba es el proceso de ejecución de un programa con la intención de encontrar errores.

Aunque esto puede sonar como un sutil juego de semántica, es en realidad una distinción importante. Entender la verdadera definición de software testing puede hacer una gran diferencia en el éxito de sus esfuerzos.

Los seres humanos tienden a ser sumamente orientados a objetivos, y el establecimiento de la meta adecuada tiene un efecto psicológico importante. Si nuestro objetivo es demostrar que un programa no tiene errores, entonces subconscientemente estaremos dirigidos hacia esta meta, es decir, tenderemos a seleccionar los datos de prueba que tienen una baja probabilidad de causar que el programa falle. Por otro lado, si nuestro objetivo es demostrar que un programa tiene errores, nuestros datos de prueba tendrán una mayor probabilidad de encontrar errores.

Este último enfoque va a añadir más valor al programa que el primero.

Un segundo problema con las definiciones tales como "La prueba es el procedimiento para demostrar que no hay errores en el programa" es que tal objetivo es imposible de lograr para prácticamente cualquier programa, incluso programas triviales.

Una vez más, los estudios psicológicos nos dicen que las personas funcionan mal cuando se proponen una tarea que ellos saben que es inviable o imposible. Por ejemplo, si le pidiéramos resolver el crucigrama de la edición dominical del New York Times en 15 minutos, probablemente observaríamos poco o ningún avance después de 10 minutos, porque la mayoría de nosotros estaríamos resignados al hecho de que la tarea parece imposible. Sin embargo, si se le pide una solución en cuatro horas, se podría esperar razonablemente ver más progresos en los 10 minutos iniciales. Definir a la prueba del software como el proceso de descubrir errores en un programa hace que sea una tarea factible, superando de este modo este problema psicológico.

Un tercer problema con las definiciones comunes, tales como "la prueba es el procedimiento para demostrar que un programa hace lo que tiene que hacer" es que los programas que hacen lo que tienen que hacer aún pueden contener errores. Es decir, es claramente un error si un programa no hace lo que se supone que debe hacer, pero los errores también están presentes si un programa hace lo que no se supone que debe hacer. Considere el programa triángulo de la sección 2.1. Incluso si pudiéramos demostrar que el programa distingue correctamente entre todos los triángulos escaleno, isósceles y equiláteros, el programa todavía tendría un error si hace algo que no se supone que debe hacer (por ejemplo, clasificar 1,2,3 como un triángulo escaleno o decir que 0,0,0 representa un triángulo equilátero). Somos más propensos a descubrir esta última clase de errores si vemos al software testing como el proceso de encontrar errores, que si lo vemos como el proceso de demostrar que un programa hace lo que se supone que debe hacer.

En resumen, es más apropiado ver al testing como el proceso destructivo de tratar de encontrar errores (cuya presencia se da por sentado) en un programa. Un caso de prueba exitoso es aquel que promueve el progreso en esta dirección, haciendo que el programa falle. Por supuesto, finalmente se desea utilizar el testing para establecer un cierto grado de confianza en que un programa hace lo que se supone que debe hacer y no hace lo que no se supone que debe hacer, pero esto se logra mejor mediante una búsqueda concienzuda de los errores.

2.4 Aspectos económicos del testing

Teniendo en cuenta la definición de software testing mencionada anteriormente, el siguiente paso es la determinación de si es posible probar un programa para encontrar todos sus errores. Le mostraremos que la respuesta es negativa, incluso para programas triviales. En general, no es práctico, y a menudo imposible, encontrar todos los errores en un programa. Este problema fundamental, a su vez, tendrá consecuencias para la economía del testing, los supuestos que el tester tendrá que hacer sobre el programa y la forma en que se diseñan los casos de prueba.

Para combatir los problemas relacionados con la economía del testing, se deben establecer algunas estrategias antes de comenzar. Dos de las estrategias más comunes incluyen las pruebas de caja negra (black-box testing) y las pruebas de caja blanca (white-box testing), que vamos a introducir en las dos secciones siguientes.

2.4.1 Pruebas de caja negra

Una estrategia de testing importante es la prueba de caja negra (black-box, data-driven, ó input/output-driven testing). Para utilizar este método, vea el programa como una caja negra. Su objetivo es despreocuparse totalmente del comportamiento interno y la estructura del programa. En su lugar, concentrarse en encontrar circunstancias en las que el programa no se comporta de acuerdo a sus especificaciones[1].

En este enfoque, los datos de prueba se derivan únicamente de las especificaciones (es decir, sin tomar ventaja del conocimiento de la estructura interna del programa).

Si desea utilizar este método para encontrar todos los errores en el programa, el criterio es la prueba exhaustiva de las entradas, generando un caso de prueba para cada de una de las condiciones de entrada posibles. ¿Por qué? Si se probaron tres casos de triángulo equilátero, para el programa de triángulo, esto de ninguna manera garantiza la detección correcta de todos los triángulos equiláteros. El programa puede contener un control especial para los valores de 3842,3842,3842 y clasificar ese triángulo como un triángulo escaleno. Siendo que el programa es una caja negra, la única manera de estar seguro de detectar la presencia de tal condición es tratar todas las condiciones de entrada.

[1] Especificaciones: Uno o más documentos en papel o formato electrónico que describen lo que debe hacer el programa.

Para probar el programa triángulo en forma exhaustiva, habría que crear casos de prueba para todos los triángulos válidos hasta el tamaño máximo del tipo entero del lenguaje de desarrollo. Esto en sí mismo es un número astronómico de casos de prueba, pero no es de ninguna manera exhaustivo, sino que aún sería necesario encontrar errores en los que el programa dijo que -3,4,5 es un triángulo escaleno y que 2, A, 2 es un triángulo isósceles. Para estar seguro de encontrar todos esos errores, hay que probar no sólo todas las entradas válidas, sino todas las entradas posibles. Por lo tanto, para probar el programa triángulo en forma exhaustiva, tendría que producir casi un número infinito de casos de prueba, que por supuesto no es posible.

Si esto suena difícil, una prueba exhaustiva de programas más grandes sería un problema incluso mayor. Considere la posibilidad de intentar una prueba de caja negra exhaustiva de un compilador[2] de C++[3]. No sólo se tienen que crear casos de prueba que representen todos los programas C++ sintácticamente válidos (de nuevo, un número prácticamente infinito), sino que tendría que crear casos de prueba para todos los programas inválidos (un número infinito) para asegurar que el compilador los detecta como no válidos. Es decir, el compilador tiene que ser probado para asegurar que no hace lo que no se supone que debe hacer (por ejemplo, compilar correctamente un programa sintácticamente incorrecto).

El problema es aún peor para los programas que tienen una memoria, como los sistemas operativos o aplicaciones con bases de datos. Por ejemplo, en una aplicación con base de datos como un sistema de reserva de vuelos, la ejecución de una operación (por ejemplo, una consulta de base de datos, o una reserva de un vuelo) depende de lo que pasó en operaciones anteriores. Por lo tanto, no sólo tendría que probar todas las operaciones válidas y no válidas individuales, sino también todas las posibles secuencias de operaciones.

Este análisis muestra que una prueba exhaustiva de las entradas es imposible.

Dos implicaciones de esto son que:

1) No se puede probar un programa para garantizar que esté libre de errores.
2) Una consideración fundamental en las pruebas del programa es la economía.

Es decir, como una prueba exhaustiva no es opción, el objetivo debe ser maximizar el rendimiento de la inversión en testing, maximizando el número de errores detectados

[2] Compilador: Herramienta que traduce el código fuente (escrito por un programador) a código de máquina (ejecutable).
[3] C++: Lenguaje de programación muy potente y orientado a objetos, basado en el lenguaje C.

usando un número finito de casos de prueba. Hacerlo implicará, entre otras cosas, ser capaz de mirar dentro del programa y hacer ciertas suposiciones razonables, pero no infalibles, sobre el programa (por ejemplo, si el programa detecta 2,2,2 como un triángulo equilátero, parece razonable que haga lo mismo para 3,3,3). Esto formará parte de la estrategia de diseño de los casos de prueba abordada en la unidad 2.

2.4.2 Pruebas de caja blanca

Otra estrategia de prueba, prueba de caja blanca (white-box ó logic-driven testing), permite examinar la estructura interna del programa. Esta estrategia deriva los datos de prueba de un análisis de la lógica del programa (y, a menudo, por desgracia, ante la omisión en la especificación).

El objetivo es establecer, para esta estrategia, la analogía a la prueba exhaustiva de las entradas del enfoque de caja negra. Hacer que cada línea en el programa se ejecute al menos una vez puede parecer la respuesta, pero no es difícil demostrar que esto es muy insuficiente.

Sin machacar el tema, ya que esta cuestión se analiza con más detalle en la unidad 2, generalmente se considera que esta analogía es la prueba de ruta exhaustiva (exhaustive path testing). Es decir, si se ejecutan mediante de casos de prueba todos los posibles caminos (rutas) del flujo de control, entonces, posiblemente, el programa ha sido completamente probado.

Sin embargo, hay dos errores en esta afirmación. Uno de ellos es que el número de trayectorias lógicas únicas a través de un programa podría ser astronómicamente grande. Para ver esto, considere el programa trivial representado en la Figura 2-1. El diagrama es un grafo de flujo de control. Cada nodo (círculo) representa un bloque de instrucciones que se ejecutan secuencialmente, posiblemente terminando con una instrucción de ramificación. Cada arista o arco representa una transferencia de control (rama) entre bloques. El diagrama representa un programa de 10 ó 20 líneas, consistente en un ciclo DO que itera 20 veces. Dentro del cuerpo del ciclo DO hay un conjunto instrucciones IF anidadas. Determinar el número de rutas lógicas únicas es lo mismo que determinar el número total de formas únicas de moverse del punto *a* al punto *b* (suponiendo que todas las decisiones en el programa son independientes entre sí). Este número es de aproximadamente 10^{14} ó 100 billones. Se calcula como 5^{20}, donde 5 es el número de trayectorias a través del cuerpo del ciclo. Como la mayoría de las personas tienen dificultades para visualizar un número tan grande, véalo de esta manera: Si usted pudiera

escribir, ejecutar y verificar un caso de prueba cada cinco minutos, se necesitarían aproximadamente mil millones de años para probar todos los casos.

Si usted fuera 300 veces más rápido, completando una prueba por segundo, podría completar el trabajo en 3 millones de años más o menos.

Por supuesto, en los programas reales cada decisión no es independiente de las demás, lo que significa que el número de posibles rutas de ejecución sería algo menor. Por otro lado, los programas reales son mucho más grandes que el simple programa representado en la Figura 2-1. Por lo tanto, las pruebas de ruta exhaustiva, al igual que las pruebas de entrada exhaustiva, parecen ser poco prácticas, si no imposibles.

El segundo error en la afirmación "la prueba de ruta exhaustiva garantiza una prueba completa" es que si todos los caminos en un programa pudieran ser probados, aun así el programa podría estar repleto de errores. Hay tres explicaciones para esto.

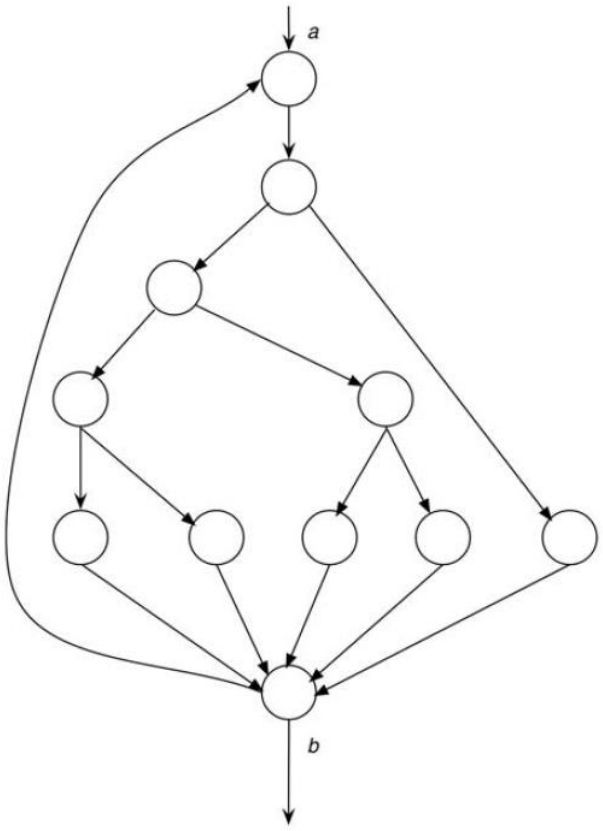


Figura 2-1. Grafo de flujo de control de un programa pequeño

La primera es que una prueba de ruta exhaustiva de ninguna manera garantiza que el programa respeta su especificación. Por ejemplo, si se le pide que escriba una rutina de ordenamiento ascendente, pero por error produjo una rutina de ordenamiento descendente, la prueba de ruta exhaustiva sería de poco valor, el programa todavía tiene un error: es el programa equivocado, no cumple con la especificación.

En segundo lugar, un programa puede ser incorrecto debido a caminos faltantes. La prueba de ruta exhaustiva, por supuesto, no podría detectar la ausencia de rutas necesarias.

En tercer lugar, una prueba de ruta exhaustiva no puede descubrir errores sensibles a los datos. Hay muchos ejemplos de este tipo de errores, pero un ejemplo sencillo debería ser suficiente. Supongamos un programa que tiene que comparar la convergencia de dos números, es decir, ver si la diferencia entre los dos números es menor que un cierto valor predeterminado.

Por ejemplo, podría escribir en lenguaje Java la siguiente instrucción IF[4]:

```
if (a-b < c)
      System.out.println("La diferencia es menor.");
```

Por supuesto, la instrucción contiene un error, ya que debería comparar c con el valor absoluto[5] de a-b. La detección de este error, sin embargo, depende de los valores usados para a y b, y no necesariamente sería detectado con sólo ejecutar cada camino a través del programa.

En conclusión, aunque la prueba exhaustiva de las entradas es superior a la prueba de ruta exhaustiva, ninguna resulta ser útil porque ambas son inviables.

Quizás, entonces, haya maneras de combinar elementos de la prueba de caja negra y caja blanca para obtener una estrategia de prueba razonable, pero no infalible. Este tema se desarrolla en la unidad 2.

[4] If (si en inglés) Estructura que permite ejecutar una o más instrucciones sólo si se cumple una condición.
[5] El valor absoluto o módulo de un número es su valor numérico sin tener en cuenta su signo. Por ejemplo, 3 es el valor absoluto de +3 y de -3.

2.5 Objetivos y limitaciones del testing

De lo explicado en las secciones anteriores se deprende que como no se pueden probar todas las combinaciones para demostrar que el programa es libre de errores, el objetivo es maximizar el rendimiento del test maximizando la cantidad de errores descubiertos dada una cierta cantidad de recursos invertidos en el test. Para esto se pueden utilizar pruebas caja negra haciendo suposiciones sobre los posibles resultados (por ejemplo, que si 2,2,2 es clasificado correctamente como triángulo equilátero entonces 3,3,3 también lo sería), o usar pruebas de caja blanca eligiendo los casos de prueba de acuerdo a la lógica del programa (aunque vimos que esto tampoco garantiza una detección de todos los errores).

Por último quisiéramos recalcar los siguientes tres conceptos en relación con el objetivo del testing:

- La prueba es el proceso de ejecución de un programa con la intención de encontrar errores.
- Un buen caso de prueba es uno que tiene una alta probabilidad de detectar un error aún no descubierto.
- Un caso de prueba exitoso es aquel que detecta un error aún no descubierto.

Con respecto a las limitaciones mencionadas sobre los casos de prueba a utilizar se deben definir prioridades [Eve07] para optimizar el rendimiento del testing.

2.5.1 Primera prioridad en el testing

Se debe hacer de los mayores riesgos de negocio (business killers) la primera prioridad del testing.

Cuando se enfrenta con personal de testing limitado, herramientas de prueba limitadas y escaso tiempo para completar las pruebas (como en la mayoría de los proyectos), es importante asegurarse de que hay suficientes recursos para hacer frente a las pruebas de al menos los riesgos de negocio principales. Cuando los recursos de pruebas no pueden cubrir los mayores riesgos de negocios, proceder con las pruebas de todos modos les daría a las personas que interactúan con el sistema la falsa expectativa de que la compañía está a salvo de los defectos del software.

2.5.2 Segunda Prioridad: Regla del 80/20

Una vez que tenga los mayores riesgos del negocio bien bajo control, considere como segunda prioridad las actividades más frecuentes. Es de conocimiento común en la industria que el 80% de toda la actividad empresarial diaria es provista por un 20% de las funciones del sistema, transacciones o workflows de negocio. Esto se conoce como la regla del 80/20.

Así que debe concentrar las pruebas en el 20% que realmente impulsa el negocio. Debido a que la escasez de recursos de testing sigue siendo una preocupación, este enfoque proporciona una manera de "llevarse más por el mismo precio". El otro 80% del sistema por lo general representa las transacciones excepcionales que se invocan sólo cuando el 20% más activo no puede resolver un problema. Una excepción a este enfoque es una actividad de negocios que se produce muy rara vez, pero la importancia las pruebas va mucho más allá de lo sugerido por su frecuencia de uso. El ejemplo clásico de una actividad de negocios “durmiente” es el cierre de fin de año para un sistema financiero.

2.6 Un enfoque estratégico

El testing de software es un conjunto de actividades que se pueden planear anticipadamente y ejecutar como un proceso sistemático. Por esta razón se debe definir un modelo para el proceso de software (un conjunto de pasos en los que se pueden incluir técnicas específicas de diseño de casos de prueba y métodos de testing).

En la bibliografía se han propuesto varias estrategias de testing, todas las cuales tienen las siguientes características generales:

- Para realizar un testing efectivo se deben realizar revisiones técnicas efectivas. Mediante esto se eliminan muchos errores antes de comenzar el testing.
- El testing comienza a nivel componente y funciona “hacia afuera”, hacia la integración del sistema completo.
- Distintas técnicas de testing son apropiadas para distintos enfoques de ingeniería de software y en distintos momentos.

- El testing lo dirige el desarrollador de software y (para proyectos medianos a grandes) un grupo de testers independiente.
- Testing y depuración (debugging) son actividades independientes, pero el debugging debe estar contemplado en toda estrategia de testing.

Una estrategia de testing debe contemplar tests de bajo nivel, necesarios para verificar que un pequeño fragmento de código fuente fue implementado correctamente, así como también tests de alto nivel para validar que la funcionalidad del sistema responde a los requerimientos del cliente.

Una estrategia debe proveer una guía para el ejecutor y un conjunto de hitos (milestones) para el gerente de proyecto. Debido a que los pasos de la estrategia de testing se ejecutan en un momento en que la presión de los tiempos (deadlines) comienza a aumentar, el progreso debe ser medible y los problemas deben descubrirse lo antes posible.

2.6.1 Verificación y validación

La prueba del software es un elemento de un concepto más amplio que suele denominarse verificación y validación (VyV). Verificación es el conjunto de actividades que aseguran que el software implemente correctamente una función específica. Validación es un conjunto diferente de actividades que aseguran que el software construido respeta los requisitos del cliente (requerimientos). Boehm [Boe81] lo establece de otra manera:

- Verificación: "¿Estamos construyendo el producto correctamente?"
- Validación: "¿Estamos construyendo el producto correcto?"

La verificación y la validación abarcan una amplia lista de actividades de aseguramiento de la calidad del software [Pre10]: revisiones técnicas formales, auditorías de calidad y de configuración, monitoreo del desempeño, simulación, factibilidad, revisión de la documentación y la base de datos, análisis de algoritmos, pruebas de desarrollo, de facilidad de uso, de calificación y de instalación. Aunque las actividades de prueba tienen un papel demasiado importante en VyV, también se necesitan muchas otras actividades.

Las pruebas son el último bastión para la evaluación de la calidad y, de manera más pragmática, el descubrimiento de errores. Pero las pruebas no deben considerarse como una “red de seguridad”. Como suele decirse: “No es posible probar la calidad. Si no está ahí antes de que empiece la prueba, no estará cuándo se termine”. La calidad se

incorpora al software en todo el proceso de ingeniería. La aplicación correcta de métodos y herramientas, las revisiones técnicas formales y efectivas junto con una administración y una medición sólidas aportan la calidad, que se confirma durante las pruebas.

2.6.2 Roles durante el testing

En cualquier proyecto de software se presenta un conflicto de intereses cuando comienzan las pruebas. Ahora se pide a las personas que han construido el software que lo prueben. En sí, esto parece inofensivo; después de todo, ¿quién conoce mejor un programa que la persona que lo desarrolló? Por desgracia, a esos mismos desarrolladores les interesa mucho demostrar que el programa está libre de errores, que funciona de acuerdo con los requisitos del cliente y que se completará a tiempo y sin exceder el presupuesto. Cada uno de estos intereses mina las bondades de la 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 analiza, modela y luego crea un programa de computadora, junto con su documentación. Como cualquier constructor, el ingeniero del software se sentirá orgulloso del edificio que acaba de construir y mirará con recelo a cualquiera que pretenda echarlo abajo. Cuando comienza la prueba hay un intento sutil, pero definitivo, de "romper" lo que ha construido el ingeniero del software. Al ponerse en los zapatos del desarrollador la prueba parecerá (psicológicamente) destructiva. De modo que el desarrollador actuará con cuidado, diseñando y ejecutando pruebas que demostrarán el buen funcionamiento del programa en lugar de descubrir errores. Por desgracia, los errores seguirán presentes. Y si el ingeniero del software no los encuentra, ¡el cliente sí lo hará!

De las consideraciones precedentes suelen inferirse erróneamente varias malas interpretaciones: 1) que el responsable del desarrollo no debería participar en el proceso de prueba, 2) que el software debe ponerse a salvo de extraños que lo prueben sin misericordia, y 3) que quienes aplican las pruebas sólo deben participar en el proyecto cuando vayan a darse los primeros pasos de esas pruebas. Todas estas afirmaciones son incorrectas.

El desarrollador del software siempre será el responsable de probar las unidades individuales (componentes) del programa y asegurar que cada una realice la función o muestre el comportamiento para el que se diseñó. En muchos casos, el desarrollador también aplica la prueba de integración (un paso que lleva a la construcción, y la prueba,

de toda la arquitectura del software). Sólo después de que la arquitectura del software esté completa participará un grupo independiente de prueba.

El papel del grupo independiente de prueba (GIP) consiste en eliminar los problemas propios de dejar que el constructor (desarrollador) pruebe lo que él mismo ha construido. La prueba independiente elimina el conflicto de intereses que, de otra manera, estaría presente. Después de todo, al personal del GIP se le paga para que encuentre errores.

Sin embargo, el ingeniero del software no debe simplemente entregar el programa al GIP y alejarse. El desarrollador y el GIP deben trabajar unidos en todo el proyecto de software para asegurar la realización de pruebas rigurosas. Mientras éstas se realizan, el desarrollador debe estar disponible para corregir los errores que se descubran.

El GIP es parte del equipo del proyecto de desarrollo del software, ya que participa en el análisis y diseño, y además sigue participando (al planear y especificar procedimientos de prueba) en todos los pasos de un proyecto grande. Sin embargo, en muchos casos el GIP informa a la organización de aseguramiento de calidad del software, por lo que obtiene un grado de independencia que sería imposible si fuera parte de la organización encargada de la ingeniería del software.

2.6.3 Prueba unitaria

La prueba unitaria (o prueba de unidad) se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño del software: el componente o módulo de software[6]. Tomando como guía la descripción del diseño al nivel de componentes, se prueban los caminos de control importantes para descubrir errores dentro de los límites del módulo. El alcance restringido que se ha determinado para las pruebas de unidad limita la relativa complejidad de las pruebas y los errores que éstas descubren. Las pruebas de unidad se concentran en la lógica del procesamiento interno y en las estructuras de datos dentro de los límites de un componente. Este tipo de prueba se puede aplicar en paralelo a varios componentes.

La interfaz del módulo se prueba para asegurar que la información fluye apropiadamente hacia dentro y hacia fuera de la unidad de programa sujeta a prueba. Se examinan las estructuras de datos locales para asegurar que los datos temporales mantienen la integridad durante todos los pasos de la ejecución de un algoritmo. Se recorren todos los

[6] Los componentes o módulos son cada una de las partes que forman el sistema, similarmente a las piezas de un rompecabezas.

caminos independientes (basis paths) en toda la estructura para asegurar que todas las instrucciones de un módulo se hayan ejecutado por lo menos una vez. Se prueban las condiciones límite para asegurar que el módulo opera apropiadamente en los límites establecidos para restringir el procesamiento. Y, por último, se prueban todos los caminos de manejo de errores.

Es necesario probar el flujo de datos en la interfaz del módulo antes de iniciar cualquier otra prueba. Si los datos no entran ni salen apropiadamente, todas las demás pruebas resultarán inútiles. Además, durante la prueba de unidad deben recorrerse las estructuras de datos locales y evaluarse (si es posible) el impacto local sobre los datos globales.

Durante la prueba de unidad, una tarea esencial es la prueba selectiva de las rutas de ejecución. Se deben diseñar casos de prueba para descubrir errores debidos a cálculos incorrectos, comparaciones erróneas o flujos de control inapropiados. Entre los errores más comunes en los cálculos se encuentran los siguientes: aplicación incorrecta o mal entendida de la precedencia aritmética, inicialización incorrecta, falta de precisión, y representación simbólica incorrecta de una expresión. La comparación y el flujo de control están estrechamente acoplados entre sí (es decir, el flujo cambia generalmente después de una comparación). Los casos de prueba deben descubrir errores como: 1) comparaciones entre diferentes tipos de datos, 2) operadores lógicos o precedencia de éstos aplicados incorrectamente, 3) expectativa de igualdad cuando los errores de precisión hacen que sea poco probable, 4) comparación incorrecta de variables, 5) terminación inapropiada o inexistente de ciclos, 6) falla en la salida cuando se encuentra una iteración divergente, y 7) variables de ciclo modificadas de manera inapropiada.

La prueba de límites es una de las tareas más importantes de la prueba de unidad. Con frecuencia, el software falla en sus límites. Es decir, a menudo los errores ocurren cuando se procesa el enésimo elemento de una matriz de n dimensiones, cuando se evoca la i-ésima repetición de un ciclo con i pasos, cuando se encuentra el valor máximo o mínimo permisible. Es muy probable descubrir errores en los casos de prueba que se ejercen sobre la estructura de datos, el flujo de control y los valores de datos ubicados apenas debajo de los máximos o mínimos, exactamente en éstos y apenas encima de ellos.

Un buen diseño impone que se prevean las condiciones de error y que se configuren rutas de manejo de errores para modificar la ruta o terminar limpiamente el procesamiento cuando ocurra un error.

Entre los posibles errores que deben probarse cuando se evalúe el manejo de errores se encuentran los siguientes: 1) la descripción del error no se entiende, 2) el error indicado

no corresponde al encontrado, 3) la condición de error causa la intervención del sistema operativo antes de que se dispare el manejo de errores, 4) el procesamiento de la condición de excepción es incorrecto, 5) la descripción del error no proporciona información suficiente para ayudar a localizar la causa del error.

2.6.3.1 Procedimientos de prueba unitaria

La prueba de unidad suele considerarse contigua al paso de la codificación. El diseño de las pruebas de unidad puede realizarse antes de que empiece la codificación (un enfoque ágil que suele preferirse y que se describirá en la Unidad 5) o después de que se ha generado el código fuente. Una revisión de la información del diseño proporciona una guía para establecer casos de prueba que probablemente descubrirán errores en cada una de las categorías analizadas. Cada caso de prueba debe relacionarse con un conjunto de resultados esperados.

Debido a que un componente no es un programa independiente, para cada prueba de unidad se debe desarrollar software controlador (driver), de respaldo o representante (stub), o de ambos tipos. En la Figura 2-2 se ilustra el ambiente para la prueba de unidad. En casi todas las aplicaciones, un controlador no es más que un "programa principal" que acepta los datos del caso de prueba, pasa estos datos al componente (que habrá de probarse) e imprime los resultados importantes. Los respaldos sirven para reemplazar módulos subordinados al componente que habrá de probarse (o llamados por éste). Un respaldo o "subprograma simulado" usa la misma interfaz[7] que el módulo subordinado, puede realizar una mínima manipulación de datos, proporciona verificación de la entrada y devuelve el control al módulo de prueba.

Controladores y respaldos representan una sobrecarga de trabajo (overhead). Es decir, resulta necesario escribir ambos tipos de software (sin que suela aplicarse un diseño formal), pero no se entregan con el producto de software final. Si se les mantiene en un nivel simple, la sobrecarga real es relativamente pequeña. Por desgracia no es posible aplicar adecuadamente una prueba de unidad a muchos componentes con un "simple" software de sobrecarga (controladores + respaldos). En muchos casos es posible posponer la prueba completa hasta la etapa de prueba de integración (donde también se utilizan controladores o respaldos).

[7] Interfaz de comunicación. Reglas de comunicación entre dos partes.

La prueba de unidad se simplifica cuando se diseña un componente con cohesión[8] elevada. Cuando un componente atiende una única función, el número de casos de prueba se reduce y es más fácil predecir y corregir los errores.

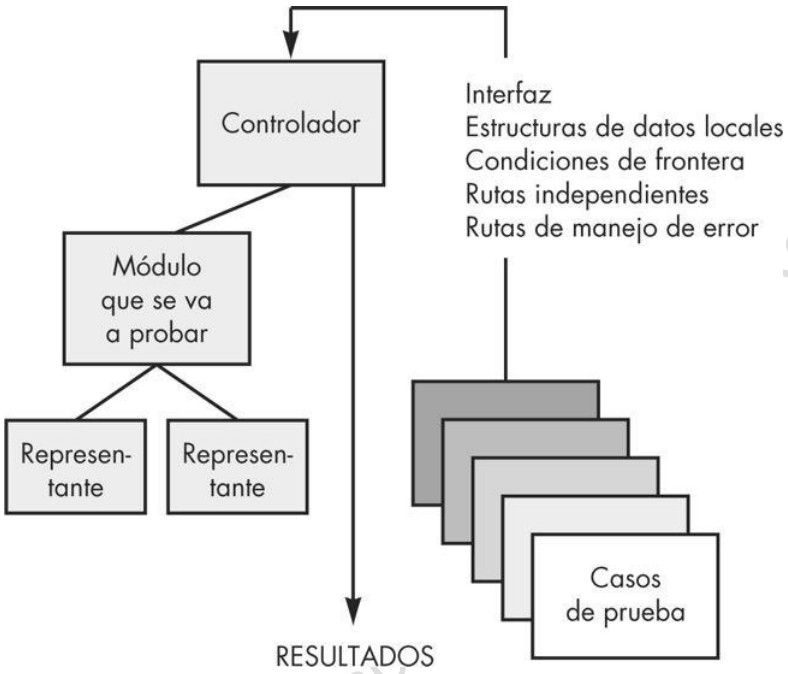


Figura 2-2. Ambiente de prueba unitaria

2.6.4 Prueba de integración

Un principiante en el mundo del software podría plantear una pregunta aparentemente legítima, una vez que se haya aplicado una prueba de unidad a todos los módulos "Si todo funciona bien individualmente, ¿por qué dudan que funcione cuando se une?"

El problema, por supuesto, consiste en "unir" (crear la interfaz). En una interfaz es posible perder datos, un módulo podría tener un efecto adverso e inadvertido sobre otro, la combinación de sub-funciones tal vez no produzca la función principal deseada, la imprecisión aceptable en elementos individuales podría ampliarse hasta grados inaceptables y las estructuras globales de datos podrían presentar problemas. Es triste, pero la lista sigue y sigue.

La prueba de integración es una técnica sistemática para construir la arquitectura del software mientras, al mismo tiempo, se aplican las pruebas para descubrir errores

[8] La cohesión tiene que ver con que cada módulo del sistema se refiera a un único proceso o entidad.

asociados con la interfaz. El objetivo es tomar componentes a los que se aplicó una prueba de unidad y construir la estructura de programa que determine el diseño.

A menudo se tiende a intentar una integración que no sea incremental; es decir, a construir el programa mediante un enfoque de "big bang". Se combinan todos los componentes por anticipado. Se prueba todo el programa como un todo. ¡Y se produce el caos! Se encuentra una gran cantidad de errores. La corrección es difícil, porque resulta complicado aislar las causas debido a la extensión del programa completo. Una vez corregidos esos errores, aparecen otros nuevos y el proceso continúa en un ciclo que parece interminable.

La integración incremental es la antítesis del enfoque del "big bang". El programa se construye y prueba en pequeños incrementos, en los cuales resulta más fácil aislar y corregir los errores, es más probable que se prueben por completo las interfaces y se vuelve factible la aplicación de un enfoque de prueba sistemática. En los siguientes párrafos se expondrán varias estrategias diferentes de integración incremental.

2.6.4.1 Integración descendente (dop-down)

La prueba de integración descendente es un enfoque incremental para la construcción de la arquitectura del software. Los módulos se integran al descender por la jerarquía de control, empezando con el módulo de control principal (programa principal). Los módulos subordinados al módulo de control principal se incorporan en la estructura de una de dos maneras: primero-en-profundidad o primero-en-anchura.

Tomando como referencia la Figura 2-3, la integración primero-en-profundidad integra todos los módulos de una ruta de control principal de la estructura del programa. La selección de una ruta principal es un poco arbitraria y depende de las características específicas de la aplicación. Por ejemplo, si se elige el camino de la izquierda, se integrarían primero los módulos M_1 M_2, y M_5. A continuación, se integraría M_8 o (si es necesario para el adecuado funcionamiento de M_2) M_6. Enseguida se construyen las rutas de control central y de la derecha. La integración primero-en-anchura incorpora todos los componentes directamente subordinados en cada nivel, desplazándose horizontalmente por la estructura. En el caso de la figura, se integrarían primero los componentes M_2, M_3 y M_4. Y les seguirían M_5, M_6, etc. El proceso de integración se realiza en una serie de cinco pasos:

1. El módulo de control principal se utiliza como controlador de prueba, y se sustituyen los respaldos en todos los componentes directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración seleccionado (es decir, primero-en-profundidad o primero-en-anchura) se van reemplazando los respaldos subordinados, uno por uno, con los componentes reales.
3. Se aplican pruebas cada vez que se integra un nuevo módulo.
4. Al completar cada conjunto de pruebas, se reemplaza otro respaldo con un módulo real.
5. Se aplica la prueba de regresión (que se analiza en la unidad 5) para asegurarse de que no se han introducido nuevos errores.

El proceso continúa a partir del paso 2 hasta la construcción total de la estructura del software.

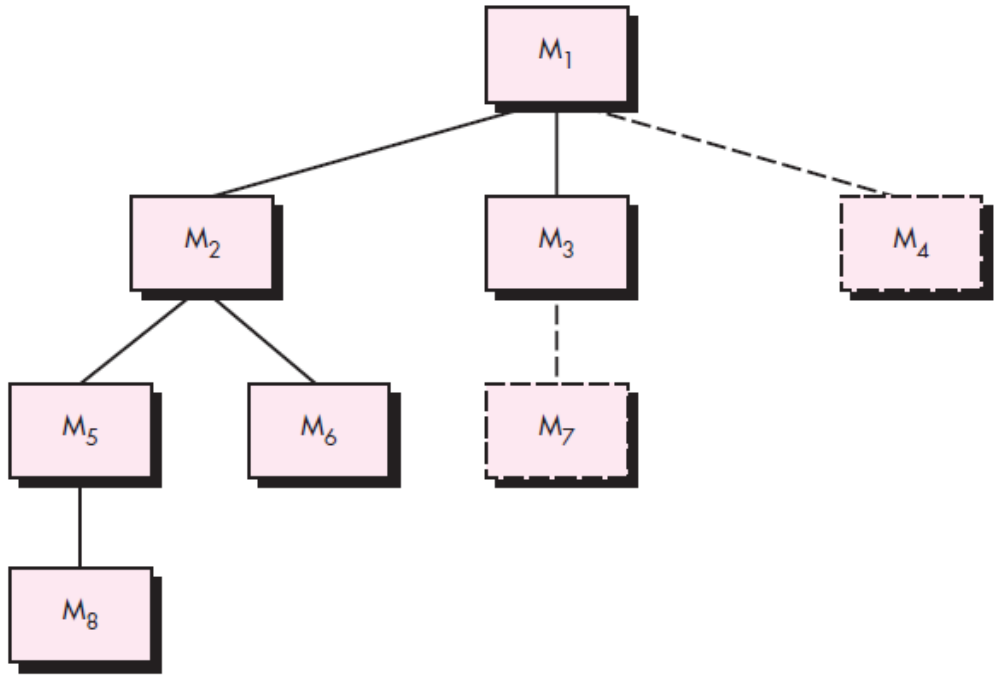


Figura 2-3. Integración descendente

2.6.4.2 Integración ascendente (bottom-up)

La prueba de integración ascendente, como su nombre lo indica, empieza la construcción y la prueba con módulos atómicos (es decir, componentes de los niveles más bajos de la

estructura del programa). Debido a que los componentes se integran de abajo hacia arriba, siempre está disponible el procesamiento requerido para los componentes subordinados a un determinado nivel y se elimina la necesidad de respaldos. Una estrategia de integración ascendente se implementa mediante los siguientes pasos:

1. Se combinan los módulos de bajo nivel en grupos (clusters) -también llamados construcciones (builds)- que realicen una sub-función específica del software.
2. Se escribe un controlador (un programa de control para pruebas) con el fin de 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 ascendiendo por la estructura del programa.

La integración sigue el patrón ilustrado en la Figura 2-4. Los componentes se combinan para formar los grupos 1, 2 y 3. Cada uno de ellos se prueba empleando un controlador (mostrado como un recuadro punteado). Los componentes de los grupos 1 y 2 están subordinados a M_a. Los controladores D_1 y D_2 se eliminan y los grupos interactúan directamente con M_a. De igual manera, se elimina el controlador D_3 del grupo 3 antes de la integración con el módulo M_b. M_a y M_b se integrarán finalmente con el módulo M_c y así sucesivamente.

A medida que la integración asciende se reduce la necesidad de controladores de prueba separados. En realidad, si los dos niveles superiores de la estructura del programa se integran de manera descendente, se reducirá de manera importante el número de controladores y se simplificará enormemente la integración de grupos.

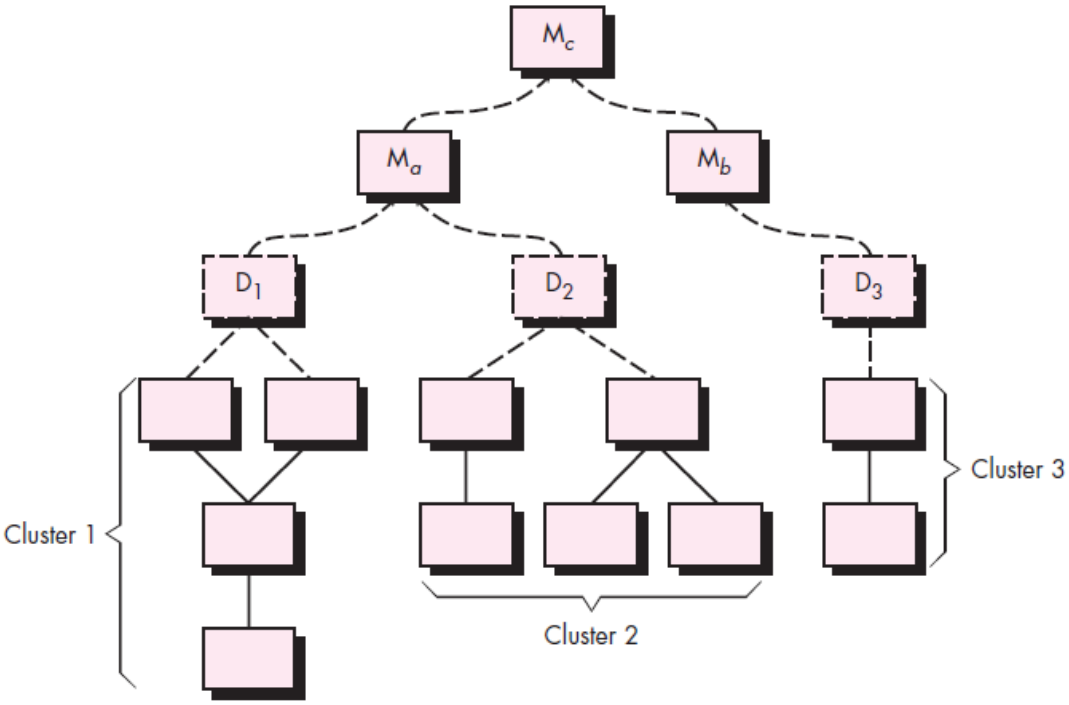


Figura 2-4. Integración ascendente

2.6.5 Prueba de validación

Las pruebas de validación empiezan tras la culminación de la prueba de integración cuando se han ejercitado los componentes individuales, se ha terminado de ensamblar el software como paquete y se han descubierto y corregido los errores de interfaz. En el nivel de validación o sistema desaparece la distinción entre software convencional y orientado a objetos. La prueba se concentra en las acciones visibles por el usuario y en la salida del sistema que éste puede reconocer.

La validación se define de muchas formas, pero una definición simple (aunque vulgar) es que se alcanza cuando el software funciona de tal manera que satisface expectativas razonables del cliente. En este punto, un desarrollador de software experimentado protestaría: "¿Qué o quién decide lo que es una expectativa razonable?"

Las expectativas razonables se definen en la especificación de requerimientos del software (un documento que describe los atributos del software visibles para el usuario). La especificación contiene una sección denominada criterios de validación. La información contenida en esa sección integra la base del enfoque de la prueba de validación.

2.6.6 Prueba de sistema

El software sólo es un elemento de un sistema de cómputo más grande. Al final, el software se incorpora a elementos del sistema (como hardware, personas, información), y se realiza una serie de pruebas de integración del sistema y de validación. Estas pruebas están más allá del alcance del proceso del software y no las realizan únicamente los ingenieros de software. Sin embargo, los pasos dados durante el diseño y la prueba del software mejorarán en gran medida la probabilidad de tener éxito en la integración del software en el sistema mayor.

Un problema clásico de la prueba del sistema es "señalar con el dedo". Esto ocurre cuando se descubre un error y el desarrollador de cada elemento del sistema culpa a los demás. En lugar de caer en este absurdo, el ingeniero del software debe anticiparse a posibles problemas con la interfaz y 1) diseñar rutinas de manejo de errores que prueben toda la información proveniente de otros elementos del sistema, 2) aplicar una serie de pruebas que simulen datos incorrectos u otros posibles errores en la interfaz del software, 3) registrar los resultados de las pruebas como "evidencia" en el caso de que se le culpe, y 4) participar en la planeación y el diseño de pruebas del sistema para asegurar que el software se ha probado adecuadamente.

En realidad, la prueba del sistema abarca una serie de pruebas diferentes cuyo propósito principal es ejercitar profundamente el sistema de cómputo. Aunque cada prueba tiene un propósito diferente, todas trabajan para verificar que se hayan integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. Las siguientes son las principales actividades que integran la prueba de sistema [Pre10]:

- Prueba de recuperación (recovery)
- Prueba de seguridad
- Prueba de resistencia (stress)
- Prueba de desempeño (performance)
- Prueba de despliegue (deployment)

2.6.7 Depuración (debugging)

La depuración ocurre como consecuencia de una prueba realizada con éxito. Es decir, cuando un caso de prueba descubre un error, la depuración es la acción que lo elimina, Aunque la depuración puede y debe ser un proceso ordenado, sigue siendo un arte. Un

ingeniero de software, al evaluar los resultados de una prueba, suele enfrentarse con una indicación "sintomática" de un problema de software. Es decir tal vez la manifestación externa del error y su causa interna no tienen una relación obvia. La depuración es el proceso mental que conecta un síntoma con una causa. Existen varias tácticas de depuración que se resumirán a continuación.

Cabe aclarar que la actividad de depuración la realiza el desarrollador y no es parte del testing. Sin embargo, es conveniente que la persona que trabaja en testing sepa de su existencia y significado aunque no se ocupe de realizarla.

2.6.7.1 Depuración por fuerza bruta

La categoría de depuración por fuerza bruta tal vez sea el método más común y menos eficiente para aislar la causa de un error del software. Los métodos de depuración por la fuerza bruta se aplican (o deberían aplicarse) cuando todo lo demás falla. Al aplicar una filosofía de "dejemos que la computadora encuentre el error", se hacen descargas de memoria (memory dumps), se invocan señales en tiempo de ejecución y se carga el programa con instrucciones de salida (impresión). En algún lugar del mar de información que se produce se espera encontrar una pista que pueda conducir a la causa de un error. Aunque la gran cantidad de información producida conduzca finalmente al éxito, lo más frecuente es que haga desperdiciar tiempo y esfuerzo.

2.6.7.2 Rastreo hacia atrás (back-tracking)

El rastreo hacia atrás es un enfoque de depuración muy común, que se utiliza con éxito en pequeños programas. Empezando en el sitio donde se ha descubierto un síntoma, se recorre hacia atrás el código fuente (manualmente) hasta hallar el lugar de la causa. Por desgracia, a medida que aumenta el número de líneas del código, la cantidad de caminos hacia atrás se vuelve tan grande que resulta inmanejable.

2.6.7.3 Eliminación de causas

El tercer enfoque para la depuración (eliminación de causas) lo determina la inducción o deducción, e introduce el concepto de partición binaria. Los datos relacionados con el error se organizan para aislar las causas posibles. Se elabora una "hipótesis de la causa"

y se aprovechan los datos ya mencionados para probar o desechar la hipótesis. Como opción, se elabora una lista de todas las causas posibles y se ejecutan pruebas para eliminar cada una de ellas. Si las pruebas iniciales indican que determinada hipótesis de causa es prometedora, se refinan los datos para tratar de aislar el error.

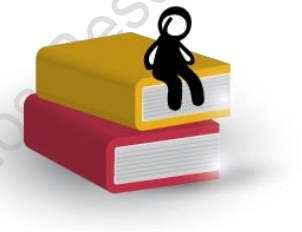
2.6.7.4 Depuración automatizada.

Cada uno de los enfoques de depuración anteriores se complementa con las herramientas de depuración que proporcionan soporte semiautomatizado al ingeniero de software mientras se intentan estrategias de depuración. Se han propuesto muchos nuevos enfoques y se dispone de muchos entornos de depuración comerciales. Los entornos de desarrollo integrado (IDEs) proporcionan una manera de capturar algunos de los errores típicos específicos del lenguaje (por ejemplo, caracteres faltantes de fin-de-instrucción, variables indefinidas, etc.) sin requerir compilación. Se cuenta con una amplia variedad de compiladores con depuración, ayudas dinámicas para la depuración ("tracers"), generadores automáticos de casos de prueba y herramientas de correlación de referencias cruzadas. Sin embargo, las herramientas no son un sustituto de una evaluación cuidadosa basada en un modelo de diseño completo y un código fuente claro.

2.6.7.5 El factor humano.

Ningún análisis de los enfoques y las herramientas de depuración estaría completo sin mencionar un poderoso aliado: ¡los demás! Un punto de vista fresco, despejado de horas de frustración, puede hacer maravillas. Una máxima final para la depuración sería: "¡Cuando todo lo demás falle, pida ayuda!"

Un análisis más detallado sobre depuración se puede consultar en [Mye04].



Bibliografía utilizada y sugerida

Libros y otros manuscritos

- [Bei90] Beizer, Boris. Software Testing Techniques. Second Edition. 1990.
- [Boe81] Boehm, B. Software Engineering Economics. 1981.
- [Cop04] Copeland, Lee. A Practitioner's Guide to Software Test Design. 2004.
- [Eve07] Everett, Gerald & McLeod, Raymond. Software Testing - Testing Across The Entire Software Development LifeCycle. 2007.
- [Far08] Farrell-Vinay, Peter. Manage Software Testing. 2008
- [Mye04] Myers, Glenford. The Art of Software Testing. Second Edition. 2004.
- [Pat05] Patton, Ron. Software Testing. Second Edition. 2005.
- [Pre10] Pressman, Roger. Software Engineering: A Practitioner's Approach. Seventh Edition. 2010.

Lo que vimos:

En esta Unidad nos adentramos en los principales problemas que surgen a la hora de realizar el testing de un proyecto de software, y en cómo un enfoque estratégico nos permite salvarlos, siempre con el objetivo de minimizar los riesgos del proyecto. En este contexto se describieron a alto nivel las etapas de esta estrategia, introduciendo para esto los conceptos necesarios.

Lo que viene:

En las próximas unidades nos enfocaremos en profundizar los conocimientos adquiridos, detallando los métodos de diseño de casos de prueba, especializándonos en el testing de tipos particulares de proyectos, y completando el conocimiento con conceptos, métodos y herramientas complementarias.