Capítulo

8

Pruebas de Software

Centro Asociado Palma de Mallorca

Tutor: Antonio Rivero Cuesta

8 Pruebas de Software.

8.1 Introducción.

A lo largo de la fase de codificación se introducen de manera inadvertida múltiples errores de todo tipo e incorrecciones respecto a las especificaciones del proyecto. Todo esto debe ser detectado y corregido antes de entregar al cliente el programa acabado.

Para garantizar la calidad del producto es necesario someter al mismo a diversas pruebas destinadas a detectar errores.

Para un software crítico, el costo de las pruebas puede ser la partida más importante del costo de todo el desarrollo.

Para evitar el caos de una prueba global única, se deben hacer pruebas por modulo, pruebas de integración y pruebas del sistema total.

8.2 Objetivos

Conocer los distintos tipos de pruebas que se deben o pueden realizar a un software para garantizar de forma razonable su correcto funcionamiento.

Se revisarán además las técnicas más utilizadas en función del tipo de lenguaje de programación.

8.3 Tipos de pruebas

Tienen un doble objetivo: la verificación y la validación.

La verificación persigue comprobar que se ha construido el producto correctamente.

La validación comprueba que el producto se ha construido de acuerdo a los requisitos del cliente.

Los procesos de verificación y validación llevan a realizar pruebas a diferentes niveles, en los que se revisa la calidad de cada etapa del proceso de elaboración del software.

Ver figura 8.1, página 284.

Tenemos pruebas de:

- Unidades.
- Integración.
- Validación.
- Sistema.

8.4 Pruebas de unidades

El principal objetivo de las pruebas es conseguir que el programa funcione incorrectamente y que se descubran sus defectos. Esto exige elaborar un juego de pruebas que someta al programa a múltiples situaciones.

Con las pruebas solo se explora una parte de todas las posibilidades del programa (Ver Figura página 286). Se pretende que con el mínimo esfuerzo posible se detecten el mayor número posible de defectos.

Para garantizar unos resultados fiables, todo el proceso de prueba se debe realizar de la manera más automática posible. Para ello, se debe crear un entorno de prueba que asegure unas condiciones predefinidas y estables para las sucesivas pasadas.

Las pruebas de unidades se realizan en un entorno de ejecución controlado, que puede ser diferente del entorno de ejecución del programa final en explotación. El entorno deberá proporcionar al menos un informe con los resultados de las pruebas y un registro de todos los errores detectados con su discrepancia respecto al valor esperado.

Se diferencian dos técnicas de prueba de unidades:

8.4.1 Pruebas de caja negra

Se ignora la estructura interna del programa y se basa exclusivamente en la comprobación de la especificación entrada- salida del software. Ver Figura Página 274.

Se trata de verificar que todos los requisitos impuestos al programa se cumplen.

Es la única estrategia que puede adoptar el cliente. El objetivo es descubrir los errores de los módulos sospechosos y para ello hay que elegir un interrogatorio amplio y coherente.

Los métodos que ayudan en la elaboración de casos de prueba son:

Partición en clases de equivalencia: se trata de dividir el espacio de ejecución del programa en varios subespacios. Cada subespacio o clase equivalente agrupa a todos aquellos datos de entrada al modulo que resultan equivalentes desde el punto de vista de la prueba de la caja negra. Ver Figura Página 288.

Por ejemplo, la equivalencia podría corresponder a que el algoritmo de cálculo, tal como se describe externamente, siga los mismos pasos en su ejecución.

Supongamos que estamos probando una función que realiza la raíz cuadrada. En este caso sería suficiente probar cualquier número positivo, el cero y cualquier número negativo. De esta forma tendríamos tres clases equivalentes. Ahora si probáramos un cuadrado perfecto positivo, un valor positivo sin raíz exacta, el cero, un cuadrado perfecto negativo y un valor negativo arbitrario, tendríamos cinco clases de equivalencia.

Hay que tener en cuenta que un caso de prueba valido para una clase puede ser también un caso de prueba invalido para otra y asimismo puede ocurrir que un caso de prueba bien elegido sea válido para varias clases.

Los pasos a seguir con este método son los siguientes:

- Definir las clases equivalentes.
- Definir una prueba que cubra tantos casos validos como sea posible de cualquier clase.
- Marcar las clases cubiertas y repetir el paso anterior hasta cubrir todos los casos validos de todas las clases.
- Definir una prueba específica para cada caso inválido. Leer Página 276.

Análisis de valores límite: hace un especial hincapié en las zonas del espacio de ejecución que están próximas al borde. Los errores en los límites pueden ser graves al solicitar recursos que no se habían preparado. Se deben proponer casos validos y casos inválidos.

Por ejemplo, si tenemos: 0<= Edad <120 años, los casos de prueba serían:

Límite Inferior : -1 0 1

Límite Superior: 119 120 121. Ver Figura Página 277.

Leer Página 278.

Comparación de versiones: se hacen diferentes versiones del mismo modulo hechas por diferentes programadores y se someten al mismo juego de pruebas. Se coparan y evalúan los resultados.

Cuando produzcan los mismos resultados podemos elegir cualquier versión. Esto no es infalible pues un error en la especificación lo arrastrarían todas las versiones.

Empleo de la intuición: la elaboración de pruebas requiere ingenio.

Las personas ajenas al desarrollo del modulo suelen aportar un punto de vista más distante y fresco.

8.4.2 Pruebas de caja transparente

Ahora se conoce y se tiene en cuenta la estructura interna del modulo. Se trata de conseguir que el programa transite por todos los posibles caminos de ejecución y ponga en juego todos los elementos del código.

Si solo efectuamos pruebas de caja negra quedaran inexplorado multitud de caminos.

Las pruebas de caja negra y caja transparente deben ser complementarias y nunca excluyentes.

Los métodos más utilizados son:

Cubrimiento lógico: se determina un conjunto de caminos básicos que recorran las líneas del flujo del diagrama alguna vez.

Cada rombo del diagrama debe representar un predicado lógico simple, es decir, no puede ser una expresión lógica con algún operador OR, AND, etc.

El número de caminos básicos necesarios vendrá determinado por el número de predicados lógicos simples que tenga el diagrama de flujo con arreglo a la siguiente fórmula:

Nº máximo de caminos = Nº de predicados + 1.

Tenemos diferentes niveles de encubrimiento:

- Nivel1: se elaboran casos de prueba para que se ejecuten al menos una vez todos los caminos básicos, cada uno de ellos por separado.
- Nivel2: se elaboran casos de prueba para que se ejecuten todas las combinaciones de caminos básicos por parejas. Ver ejemplo Página 295.
- Nivel3: Se elaboran casos de prueba un número significativo de las combinaciones posibles de caminos. Cubrir todas las combinaciones posibles resulta inabordable.

Como mínimo las pruebas de cada modulo deben garantizar el nivel 1.

Nunca se podrá detectar la falta de un fragmento de código.

Prueba de bucles: se deben elaborar pruebas para:

Bucle con número no acotado de repeticiones: ejecutar el bucle 0, 1, 2, algunas, muchas veces.

Bucle con número máximo (M) de repeticiones: ejecutar el bucle 0, 1, 2, algunas, M-1, M, M+1 veces.

Bucles anidados: ejecutar todos los bucles externos en su número mínimo de veces para probar el bucle mas interno. Para el siguiente nivel de anidamiento, ejecutar los bucles externos en su número mínimo de veces, y los bucles internos un número típico de veces. Repetir así hasta completar todos los niveles.

Bucles concatenados: si son independientes se probaran por separado con los criterios anteriores. Si están relacionados, por ejemplo, el índice final de uno es el inicial del siguiente, se empleara un enfoque similar al indicado para los bucles anidados.

Empleo de la intuición: merece la pena dedicar un cierto tiempo a elaborar pruebas que solo por intuición podemos estimar que platearan situaciones especiales.

8.4.3 Estimación de errores no detectados

Resulta imposible demostrar que un modulo no tiene defectos. Para estimar los defectos que quedan sin detectar procedemos así:

Anotar el número de errores que se producen inicialmente con el juego de casos de prueba. E_I.

Corregir el módulo hasta que no tenga ningún error con el mismo juego de casos de prueba.

Introducir aleatoriamente un número determinado de errores en los puntos más diversos del modulo. E_A.

Someter al módulo con los nuevos errores al juego de casos de prueba y contar los errores que se detectan. E_D.

Suponiendo la misma proporción, el porcentaje de errores sin detectar será al mismo para los errores iniciales que para los deliberados. Por tanto, el número estimado de errores sin detectar será:

$E_E = (E_A - E_D) * (E_I / E_D)$.

8.5 Pruebas de unidades en programación orientada a objetos.

En programación orientada a objetos debemos probar la clase como unidad, en la que las operaciones van modificando los datos encapsulados en ella y por tanto el comportamiento de la clase.

Las secuencias de pruebas de una clase se diseñan para probar las operaciones relevantes en ella y es examinando los valores de los atributos de la clase donde comprobaremos si existen errores.

Existen varios métodos para probar una clase:

- Pruebas basadas en fallo.
- Pruebas aleatorias.
- Pruebas de partición.
- En base a la capacidad de las operaciones de cambiar el estado de la clase.
- En base a los atributos que utiliza cada operación.

8.6 Estrategias de integración.

Los módulos de un producto software se han de integrar para conformar el sistema completo. En esta fase de integración también aparecen nuevos errores y se debe proceder siguiendo estrategias para facilitar la depuración de los errores que vayan surgiendo.

Entre las estrategias básicas de integración tenemos:

8.6.1 Integración Big Bang

Se realiza la integración en un único paso. En este caso la cantidad de errores que aparecen de golpe hace imposible la identificación de los defectos, provocando un caos. Solo para sistemas muy pequeños se puede justificar su utilización.

8.6.2 Integración descendente

Tenemos un modulo principal que se prueba con módulos de andamiaje o sustitutos, que mas tarde se van sustituyendo uno por uno por los verdaderos, realizando las pruebas de integración. Los sustitutos deben ser los más simples posible.

La codificación de los sustitutos es un trabajo adicional que conviene simplificar al máximo.

La ventaja es que se ven desde el principio las posibilidades de la aplicación. Esto permite mostrar muy pronto al cliente un prototipo sencillo y discutir sobre él posibles mejoras o modificaciones.

Sus inconvenientes son que limita tanto el trabajo en paralelo como el ensayo de situaciones especiales.

8.6.3 Integración ascendente

Se codifica por separado y en paralelo todos los módulos de nivel más bajo. Para probarlos se escriben módulos gestores o conductores que los hacen funcionar independientemente o en combinaciones sencillas.

Los Gestores se van sustituyendo uno a uno por los módulos de mayor nivel según se van codificando, al tiempo que se van desarrollando nuevos gestores si hace falta. El orden de sustitución puede ser cualquiera salvo para los últimos pasos en que se necesita que todos los módulos inferiores estén disponibles.

Tiene como ventajas que se facilita el trabajo en paralelo y facilita el ensayo de situaciones especiales. Su inconveniente es que es difícil ensayar el funcionamiento global hasta el final de su integración.

La mejor solución es usar integración ascendente con los módulos de nivel más bajo y descendente con los de nivel más alto, lo que se llama integración sandwich.

8.7 Pruebas de validación.

Cuando finaliza la integración y sus pruebas asociadas y el software está completamente ensamblado, nos queda validar.

La validación se consigue a través de una serie de pruebas que comprueban la conformidad con los requerimientos, además de rendimiento, ergonomía y documentación.

8.7.1 Pruebas alfa

Para comprobar que un producto software es realmente útil para sus usuarios es conveniente que estos últimos intervengan también en las pruebas finales del sistema. Es probable que los problemas más graves aparezcan ya al comienzo y por ello es aconsejable que alguien del equipo de desarrollo acompañe al usuario durante la primera toma de contacto.

8.7.2 Pruebas beta

Uno o varios usuarios trabajan con el sistema en su entorno normal, sin apoyo de nadie, anotando cualquier problema que se presente. Es muy importante que sea el usuario el encargado de transmitir al equipo de desarrollo cual ha sido el procedimiento de operación que llevó al error. Esta información resulta vital para abordar la corrección.

8.8 Pruebas de sistema.

Son pruebas al sistema completo, tenemos las siguientes clases:

Según el objetivo perseguido, tendremos diferentes clases de pruebas:

- ***Pruebas de recuperación***: sirven para comprobar la capacidad del sistema para recuperarse ante fallos.
- **Pruebas de seguridad**: sirven para comprobar los mecanismos de protección contra un acceso no autorizado.
- ***Pruebas de resistencia***: sirven para comprobar el comportamiento del sistema ante situaciones excepcionales.
- ***Pruebas de sensibilidad***: comprobar el tratamiento que da el sistema a ciertas singularidades relacionadas casi siempre con los algoritmos utilizados.
- ***Pruebas de rendimiento***: comprobar las prestaciones del sistema que son críticas en tiempo.
- ***Pruebas de despliegue o configuración***: sirven para comprobar el funcionamiento del software que debe ejecutarse en varias plataformas y sistemas operativos distintos. Permite comprobar la correcta instalación del programa y la documentación para ello.