UT06 - Pruebas del software

Autor: Luis del Saz

ÍNDICE

Índice de contenido

Versión del documento

05/06/11 Creación del documento

1 INTRODUCCIÓN

Coincidiendo con los diversos hitos durante el desarrollo de un producto software se suelen realizar controles. Durante estos controles se evalúa la calidad de la entrega y se busca detectar fallos o errores en ellas. El objetivo es detectar los fallos cuanto antes.

Las ejecuciones o ensayos posteriores a la terminación de un código se denominan **pruebas**.

Son un método más para validar y verificar el software.

La **validación** se realiza al finalizar por completo el desarrollo, para determinar si satisface los requisitos especificados. Trata de responder a la pregunta ¿Estamos construyendo el producto corrrecto?

La **verificación** se realiza al final de cada fase y se comprueba el cumplimiento de los requisitos de esas fase. Se trata de responder a la pregunta ¿Estamos construyendo el producto correctamente?

Durante el periodo de pruebas se toman un conjunto de entradas posibles o **casos de prueba** y se busca encontrar **defectos, fallo o errores** en el programa. Los defectos son procesos incorrectos que generan salidas erróneas. Los fallos son también la incapacidad para cumplir un requisito. Por ejemplo, la incapacidad de procesamiento de datos por segundo requerida.

1.1 Filosofía de las pruebas del software

El producto software tiene unas características que lo diferencian de otros productos generados en otros procesos de ingeniería. Uno de ellos es la ausencia de leyes que rijan su comportamiento y la complejidad exponencial de posibilidades que deben afrontar. Esta es una disciplina donde no se ve mal vender un producto que se sabe que falla. Es habitual que los fabricantes de los productos publiquen una lista de errores conocidos. E incluso se cobra por reparar esos errores.

La prueba exhaustiva es impracticable, incluso en programas sencillos. Un programa que sume dos números entre 1 y 100 tiene 10.000 posibles combinaciones de entradas. Para hacer una prueba exhaustiva hay que probar todas las posibles entradas.

En otras disciplinas tradicionales un buen ingeniero o trabajador comete pocos errores, por lo que se asocia la ausencia de errores a esos buenos profesionales. El hecho de encontrar un error puede hacer sentir culpable a aquel que lo cometió.

Este hecho es un problema en el desarrollo de software, porque cualquier profesional, por bueno que sea cometerá errores. Sin embargo en programación, una prueba que detecte un error no implica el fracaso del programador. En este proceso los factores que provocan la aparición del error no siempre son controlables.

1.2 Recomendaciones

Pueden hacerse una serie de recomendaciones para el proceso de pruebas:

- Para cada caso de prueba debe definirse previamente cuál es el resultado esperado. Ese resultado se comparará con el realmente obtenido en la ejecución de la prueba.

- El programador debe evitar probar sus propios programas. Es evidente que si en la fase de diseño o programación no ha tenido algo en cuenta, muy probablemente volverá a pasarlo por alto de nuevo. Además el programador, consciente o inconscientemente, deseará demostrar que su programa funciona, lo que no ayudará a encontrar los errores.
- El resultado de cada prueba debe inspeccionarse a conciencia.
- Al generar casos de prueba, hay que incluir tanto datos válidos como inválidos o inesperados.
- Hay que probar: que el software hace lo que debe hacer. Que el software no hace nada indebido. Lo es frecuente que se olvide realizarlo.
- Hay que documentar los casos de prueba.
- Asumir que siempre habrá algún defecto. EL 40% del desarrollo se emplea en pruebas y depuración.

No hay técnicas rutinarias y, al igual que en la programación, hay que recurrir casi siempre al ingenio para encontrar los fallos.

2 TÉCNICAS DE DISEÑO DE CASOS

Como se comentó antes, es imposible la prueba exhaustiva. Y esto considerando los casos válidos. Si además se consideran los inválidos la cosa se complica bastante.

Las técnicas de diseño de casos tendrán como objetivo conseguir una confianza aceptable en el producto.

Suelen elegirse para ello casos de prueba representativos, la elección aleatoria no suele dar buen resultado.

Existen dos enfoques:

1. Estructural o de caja blanca, caja transparente o caja de cristal.
2. Enfoque funcional o de caja negra.

Ambos enfoques **no son excluyentes**.

2.1 Caja blanca

El diseño de casos se basa en la elección de "caminos de ejecución" importantes que ofrezcan una seguridad aceptable de descubrir defectos. Se usan criterios de cobertura lógica para ello. Es decir, se trata de cubrir secciones del código fuente:

- **Cobertura de sentencias**: intenta que se ejecuten todas las sentencias del programa al menos una vez
- **Cobertura de decisiones**: cada decisión debe resultar con un verdadero y un falso al menos una vez.
- **Cobertura de condiciones**. Una decisión se compone de condiciones: if (a==3 && b==3) tiene dos condiciones. Se trata de que cada condición adopte sus dos posibles valores

almenos una vez.

- **Criterios de decisión/condición**: exije que se cumplan los dos puntos anteriores.
- **Cobertura de caminos**. Se define "camino" como una secuencia encadenada de sentencias desde el inicio al final de la ejecución. La **cobertura de caminos** es el criterio más elevado y consiste en ejecutar cada posible camino al menos una vez.

2.2 Caja Negra

Utiliza las siguientes reglas para generar casos de prueba:

- Cada caso debe cubrir el máximo número de valores distintos de los datos de entrada.
- Debe dividirse el conjunto de datos de entrada en clases de equivalencia de modo que la prueba de un caso representativo de una clase permita suponer razonablemente que todos los datos de esa clase no testeados van a funcionar.

Pasos:

1. Identificar las condiciones que se exigen para los datos de entrada, restricciones de formato o de valor de los mismos.
2. Tras lo anterior, separar en clases y por grupos de:
   a) Datos válidos
   b) Datos no válidos.

Veremos cómo identificar las clases de equivalencia y cómo crear esos casos de prueba.

- Si se especifica **un rango de valores** de entrada, "el número debe estar entre 1 y 49", se creará una clase válida 1<= n <= 49 y otras dos clases no válidas <1 y >49.
- Si se especifica **un número de valores**: "un piso puede tener hasta tres propietarios", se creará una clase válida "propietarios de 1 a 3". Y dos no válidas, "no hay propietarios" y "hay más de tres".
- Una situación "**x debe ser..**.", o booleana, provocará dos clases. EJ: "el CIF contiene una letra al final" genera dos clases "CIF con letra al final" y "CIF sin letra al final".
- Si hay **valores de conjunto**, "los colores de un semáforo son rojo amarillo y verde", se genera una clase válida por cada valor y una no válida con cualquier otro distinto de los anteirores.
- Si una vez creadas una clase **se sospecha que el tratamiento va a ser diferente para determinados casos**, hay que subdividir esa clase según esos casos.

La división en clases debería ser realizada por personas ajenas al desarrollo.

Una vez determinadas las clases:

1. Se asigna un número único a cada clase.

2. Se crea **un caso que cubra el mayor número posible de clases válidas** y se repite el proceso hasta cubrir todas las clases válidas.
3. Se crea **un caso para cada clase no válida**. Se hace así porque un caso erróneo podría enmascarar otros casos erróneos y no ser estos últimos reparados en la depuración posterior.

2.2.1 Análisis de valores límite

Por experiencia se ha constatado que los casos que exploran condiciones límite de un programa producen mejores resultados para detectar defectos.

Las **condiciones límite** son aquellas que se encuentran arriba, abajo y en los márgenes de las clases de equivalencia.

En lugar de elegir un representante de cada clase, se elige más de uno, probando valores cercanos a los límites, los límites mismos y valores inválidos cercanos a los límites.

Los valores límite se aplican **a valores de entrada y de salida**. Se deben escribir casos de entrada que provoquen salidas límite.

3 PRUEBAS UNITARIAS, DE INTEGRACIÓN y DE SISTEMA

3.1 Pruebas unitarias

Una unidad de prueba es uno o mas módulos que cumplen que:

- Son todos del mismo programa
- Al menos uno de ellos no ha sido probado
- El conjunto de módulos es objeto de una prueba

Las pruebas unitarias suelen diseñarse con casos de caja blanca. Las pruebas las hacen desarrolladores, pero distintos del programador del módulo.

3.2 Pruebas de integración

Presuponen que diversos módulos han sido probados ya. Tratan de probar que esos módulos interactúan y funcionan de modo correcto conjuntamente. Está pues orientado a la prueba de los interfaces (flujos de datos) entre módulos.

Suele especificarse si la prueba será total o incremental y el orden de codificación y prueba de cada módulo: desde la raíz hacia las hojas o al revés. Se detalla también el coste de las pruebas.

3.2.1 Pruebas de sistema

Se trata de probar integración hardware y software para ver si cumple los requisitos,es decir:

- Requisitos funcionales.
- Rendimiento de interfaces hardware, software, de usuario y de operador.
- Adecuación de la documentación al usuario

- Ejecución en condiciones límite y de sobrecarga.

Suelen probarse.

- Casos de caja negra aplicadas a las especificaciones.
- Casos necesarios para pruebas de rendimiento y capacidad: grandes volúmenes de datos y de transacciones. *Stress testing*
- Caja blanca en casuísticas de alto nivel, por ejemplo las especificadas en diagramas de flujo de datos de alto nivel.

3.2.2 Prueba de aceptación

El usuario realizará pruebas para evaluar la fiabilidad y facilidad de uso del software, esta prueba se llama **prueba de aceptación** y tiene como objetivo transmitir confianza al usuario en el producto y convencerle de que está listo para su uso. El usuario de be ser asesorado por un profesional, no debe probar él solo. En proyectos que reemplazan un producto antiguo, pueden establecerse periodos de funcionamiento en paralelo de ambos sistemas, para permitir comparar ambos y ver si son exactos en sus resultados.

Los criterios de aceptación estarán firmados previamente por el usuario.

En ocasiones se liberan versiones beta para prueba por usuarios de confianza distintos de los usuarios finales.

4 DOCUMENTACIÓN

Según IEEE std 829 son los siguientes:

- Plan de pruebas: plan general de esfuerzo de prueba para cada fase de la estrategia de prueba del producto.
- Se genera la especificación del diseño de la prueba (ampliación y detalle del plan de pruebas).
- Se define con detalle cada uno de los casos del apartado 2 de la UT10.
- Se debe especificar cómo ejecutar cada caso: procedimientos de prueba
- Los dos documentos anteriores son básicos para la ejecución de las pruebas.

Cada documento anterior sigue una estructura prefijada por el estándar en cuanto a apartados y contenidos.

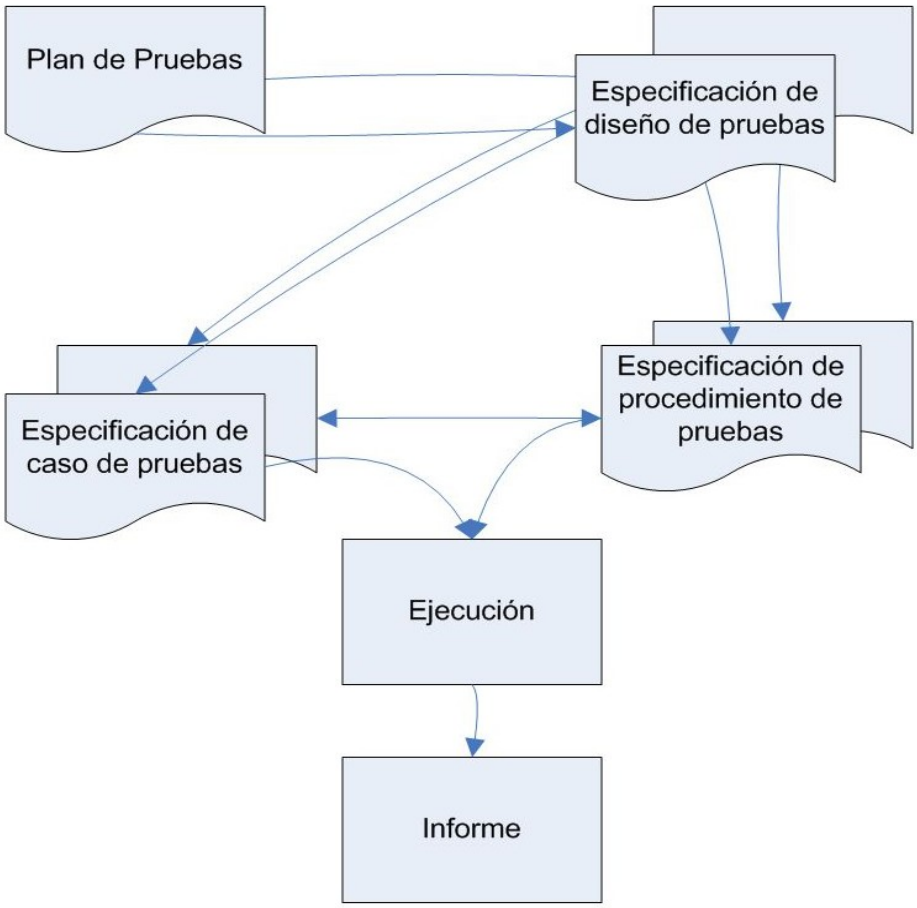

Ilustración 1: Documentos de prueba

5 BIBLIOGRAFÍA

- ✔ "Análisis y Diseño de Aplicaciones Informáticas de Gestión". RA-MA.