UNIDAD 3. DISEÑO Y REALIZACIÓN DE PRUEBAS

1 INTRODUCCIÓN

Una de las últimas fases del ciclo de vida antes de entregar un programa para su explotación, es la fase de pruebas.

Una de las sorpresas con las que se suelen encontrar los nuevos programadores es la enorme cantidad de tiempo y esfuerzo que requiere esta fase. Se estima que la mitad del esfuerzo de desarrollo de un programa (tanto en tiempo como en gastos) se va en esta fase. Si hablamos de programas que involucran vidas humanas (medicina, equipos nucleares, etc) el costo de la fase de pruebas pude fácilmente superar el 80%.

Pese a su enorme impacto en el coste desarrollo, es una fase que muchos programadores aún consideran clasificable como un arte y, por tanto, como difícilmente conceptualizable. Es muy difícil entrenar a los nuevos programadores, que aprenderán mucho más de su experiencia que de lo que les cuenten en los cursos de programación.

Una vez que tenemos nuestros algoritmos diseñados y pasados a algún lenguaje de programación, pasamos a “probarlo” ejecutándolo. En programas sencillos, con un par de ejemplos podemos asegurarnos que el programa funciona correctamente. Pero a medida que los programas se van complicando, para saber si el desarrollo es correcto no nos vale con un par de números para probarlo.

Debemos establecer todo un mecanismo de pruebas para que se nos garantice la correcta ejecución del programa en todas las condiciones posibles con todos los datos posibles de entrada, control de errores, validación de datos, etc.

1.1 ¿QUÉ ES PROBAR?

Como parte que es de un proceso industrial, la fase de pruebas añade valor al producto que se maneja: todos los programas tienen errores y la fase de pruebas los descubre; ese es el valor que añade. El objetivo específico de la fase de pruebas es encontrar cuantos más errores, mejor.

Es frecuente encontrarse con el error de afirmar que el objetivo de esta fase es convencerse de que el programa funciona bien. En realidad, ese es el objetivo propio de las fases anteriores (¿quiñen va a pasar a la sección de pruebas un producto que sospecha que está mal?). Cumplido ese objetivo, lo mejor posible, se pasa a pruebas. Esto no obsta para reconocer que el objetivo último de todo el proceso de fabricación de programas sea hacer programas que funcionen bien; pero cada fase tiene su objetivo específico, y el de las pruebas es destapar errores.

Probar un programa es ejercitarlo con la peor intención a fin de encontrarle fallos.

En los apartados siguientes vamos a presentar el diseño y la realización de pruebas pertinentes para nuestros programas.

2 PLANIFICACIÓN DE PRUEBAS

El propósito de la planificación de pruebas es especificar el alcance, enfoque, recursos requeridos, calendario, responsables y manejo de riesgos de un proceso de pruebas.

Nótese que puede haber un plan global que especifique el énfasis a realizar sobre los distintos tipos de prueba (verificación, integración y validación).

Entre los documentos relacionados con el diseño de las pruebas según estándar IEEE 829.

1. **Identificador único del documento:** Preferiblemente de alguna forma mnemónica que permita relacionarlo con su alcance, por ejemplo TP-Global (plan global del proceso de pruebas), TP-Req-Seguridad 1 (plan de verificación del requerimiento 1 de seguridad), TP-Contr_X (plan de verificación del contrato asociado al evento de sistema X), TP-Unit-Despachador.iniciar (plan de prueba unitario para el método iniciar de la clase Despachador). Como todo artefacto del desarrollo, está sujeto a control de configuración, por lo que debe distinguirse adicionalmente la versión y fecha del plan.

2. **Alcance:** Indica el tipo de prueba y las propiedades/elementos del software a ser probado.

3. **Introducción y resumen de elementos y características a probar:** Indica la configuración a probar y las condiciones mínimas que debe cumplir para comenzar a aplicarle el plan. Por un lado, es difícil y arriesgado probar una configuración que aún produce errores; por otro lado, si esperamos a que todos los módulos estén perfectos, puede que detectemos fallos graves demasiado tarde.

4. **Elementos software a probar:** Algunos elementos que se han integrado en nuestro sistema ya están probados para otros y si mantenemos las condiciones en las que su uso está establecido, podemos evitar su inclusión en las pruebas. Como mucho, aparecerán en las pruebas integración de módulos.

5. **Características para probar:** Dentro de cada elemento, debemos definir todas las características que hay que probar.

6. **Características que no se probarán:** Importante establecer también las características que no se probarán y por qué no se hará. En un repaso posterior puede que alguna de estas características sea la causante de errores, por lo que podría incluirse en las características a probar.

7. **Enfoque general de la prueba:** Establecer objetivos generales de las pruebas a realizar.

8. **Estrategia:** Describe la técnica, patrón y/o herramientas a utilizarse en el diseño de los casos de pruebas. Por ejemplo, en el caso de pruebas unitarias de un procedimiento, esta sección podría indicar: “Se aplicará la estrategia caja-negra de fronteras de la precondición” o “Ejercicio de los caminos ciclomáticos válidos”. En lo posible la estrategia debe precisar el número mínimo de casos de prueba a diseñar, por ejemplo 100% de las fronteras, 60% de los caminos ciclomáticos. La estrategia también explicita el grado de automatización que se exigirá, tanto para la generación de casos de prueba como para su ejecución.

9. **Criterios de paso/fallo para cada elemento:** Se deben establecer en qué condiciones una prueba de un elemento se considera válido y en qué casos en no válida la prueba.
10. **Criterios de suspensión y requisitos de reanudación:** Especifica las condiciones bajo las cuales, el plan debe ser:
    - Suspendido
    - Repetido
    - Finalizado
    - Reanudado
11. **Documentos para entregar:** Se refiere a los documentos en los que se recogerán todas las pruebas realizadas, sus resultados esperados, los obtenidos y un análisis de las desviaciones, así como propuestas de mejora.
12. **Actividades de preparación y ejecución de pruebas:** Indica la secuencia de actividades que se deben llevar a cabo para la preparación y ejecución de las pruebas, bien conjuntas, bien por separado.
13. **Necesidades de entorno:** Se debe especificar claramente el entorno y las necesidades que supondrá la realización de las pruebas. El entorno debe ser lo más exacto posible al entorno real en que se implementará el sistema probado.
14. **Responsabilidades en la organización y realización de las pruebas:** Especifica quién es el responsable de cada una de las tareas previstas en el plan.
15. **Necesidades de personal y formación:** Toda actividad requiere de personal cualificado y en caso necesario, se plantearán acciones formativas para que las personas encargadas de realizar las pruebas sepan en todo momento qué es lo que tienen que hacer y además que puedan responder imprevistos.

16. **Esquema de tiempos:** Esta sección describe los hitos del proceso de prueba y el grafo de dependencia en el tiempo de las tareas a realizar.
17. **Riesgos asumidos por el plan y planes de contingencias:** Especifica los riesgos del plan, las acciones mitigantes y de contingencia.
18. **Aprobaciones y firmas con nombre y puesto desempeño:** Por último, debe aparecer en todo documento relativo a las pruebas, las firmas, nombres y puestos de los responsables de la aprobación de la prueba en general o las pruebas específicas, según sea el caso.

Es importante distinguir entre Validar y Verificar:

Verificación: El proceso de evaluación de un sistema (o de uno de sus componentes) para determinar si los productos de una fase dada satisfacen las condiciones impuestas al comienzo de dicha fase. Responde a la pregunta: ¿Estamos construyendo correctamente el producto?

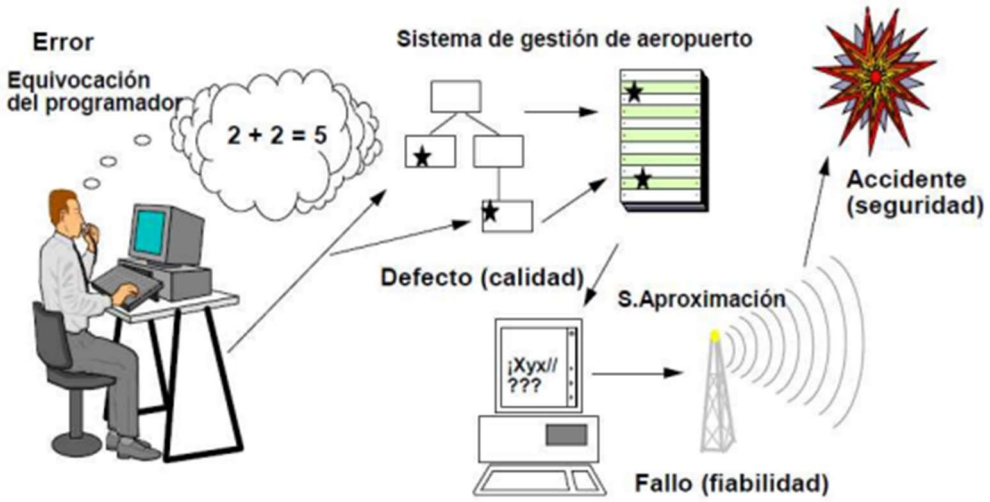
Validación: El proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos marcados por el usuario. Responde a: ¿Estamos construyendo el producto correcto?

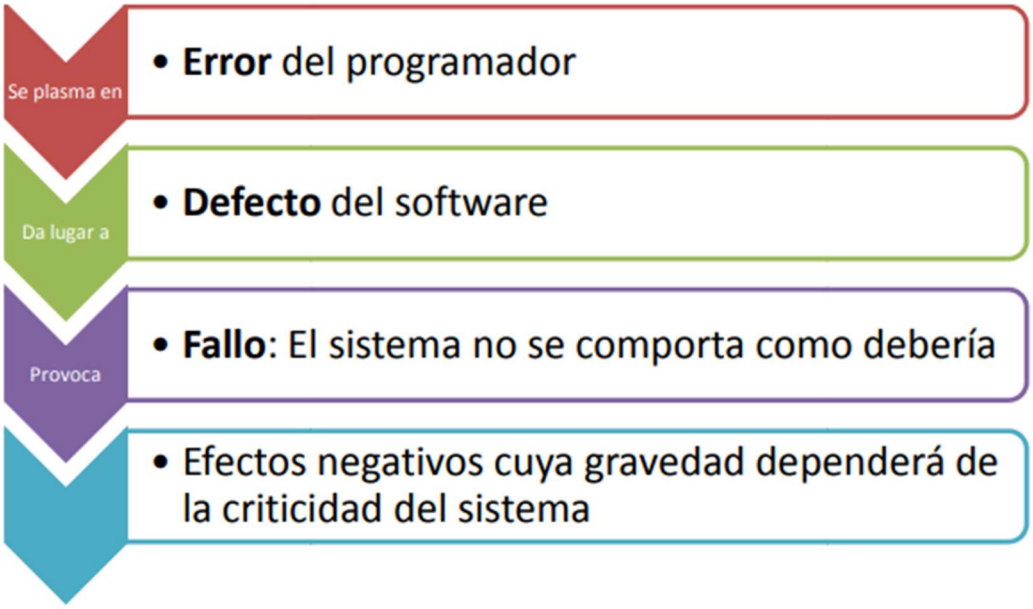
Veamos a continuación algunas definiciones:

- **Pruebas (test):** “Una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto”
- **Caso de pruebas (test case):** “Un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular”.
- **Defecto (defect, fault, “bug”):** “Un defecto en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa”.

- **Fallo (failure):** “La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados”.
- **Error (error):** Tienen varias acepciones:
    - La diferencia entre un valor calculado, observado o medio y el valor verdadero, especificando o teóricamente correcto.
    - Un defecto
    - Un resultado incorrecto
    - Una acción humana que conduce a un resultado incorrecto

Relación entre Error, Defecto y Fallo:





2.1 IDEAS PARADÓJICAS DE LAS PRUEBAS

La prueba ideal de un sistema sería exponerlo en todas las situaciones posibles, así encontraríamos hasta el último fallo. Indirectamente, garantizamos su respuesta ante cualquier caso que se le presente una la ejecución real.

Esto es imposible desde todos los puntos de vista: humano, económico e incluso matemático.

Dado que todo es finito en programación (el número de líneas de códigos, el número de variables, el número de valores en un tipo, etc) cabe pensar que el número de pruebas posibles es finito. Esto deja de ser cierto en cuanto entran en juego de bucles, en los que es fácil introducir condiciones para un funcionamiento si fin. Aún en el irrealista caso de que el número de posibilidades fuera finito, el número de combinaciones posibles es tan enorme que se hace imposible su identificación y ejecución a todos los efectos prácticos.

Probar un programa es someterles a todas las posibles variaciones de los datos de entrada, tanto si son válidos como si no lo son. Imaginase hacer esto con un compilador de cualquier lenguaje: ¡habría que escribir, compilar y ejecutar todos y cada uno de los programas que se pudieran escribir con dicho lenguaje!

Sobre esta premisa de imposibilidad de alcanzar la perfección, hay que buscar formas humanamente abordables y económicamente aceptables de encontrar errores. Nótese que todo es muy relativo y resbaladizo en esta área.

Como conclusión tenemos los siguientes puntos:

- La prueba exhaustiva del software es impracticable (no se pueden probar todas las posibilidades de su funcionamiento ni siguiera en programas sencillos.
- El objetivo de las pruebas es la detección de defectos en el software (describir un error es el éxito de una prueba)
- Mito: Un defecto implica que somos malos profesionales y que debemos sentirnos culpable → No es cierto: Todo el mundo comete errores.
- El descubrimiento de un defecto significa un éxito para la mejora de la calidad.

2.2 RECOMENDACIONES PARA UNAS PRUEBAS EXITOSAS

- Cada caso de pruebas debe definir el resultado de salida esperando que se comparara con el realmente obtenido.
- El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas.
- Además, es normal que las situaciones que olvidó considerar al crear el programa queden de nuevo olvidados al crear los casos de prueba.
- Se debe inspeccionar a conciencia el resultado de cada prueba, así, poder descubrir posibles síntomas de defectos.
- Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.
- Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):

    - Probar si el software no hace lo que debe hacer
    - Probar si el software hace lo que debe hacer, es decir, si provoca efectos secundarios adversos.

- Se deben evitar los casos desechables, es decir, los no documentados ni diseñados con cuidado: Ya que suele ser necesario probar muchas veces el software y por tanto hay que tener claro qué funciona y qué no.
- No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas → Siempre hay defectos.
- La experiencia parece indicar que donde hay un defecto hay otros, es decir, la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubierto.
- Las pruebas son una tarea tanto o más creativa que el desarrollo de software. Siempre se ha considerado las pruebas como una tarea destructiva y rutinaria.
- Es interesante planificar y diseñar las pruebas para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo.

2.3 PROCESO DE PRUEBA

ACTIVIDADES:

- La depuración (localización y corrección de defectos)
- El análisis de la estadística de errores: Sirve para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y por tanto mejorar los procesos de desarrollo.

3 TIPOS DE PRUEBAS

Hay multitud de conceptos (y palabras clave) asociadas a las tareas de prueba. Clasificarlas es difícil, pues no son mutuamente disjuntas, sino muy entrelazadas. En lo que sigue intentaremos la siguiente estructura:

Fase de prueba:

- UNIDADES
    - Planteamiento
- CAJA BLANCA
    - Cobertura:
        - De segmentos
        - De ramas
        - De condición/decisión
        - De bucles
- CAJA NEGRA
    - Cobertura de requisitos
- INTEGRACIÓN
- ACEPTACIÓN

La prueba de unidades se plantea a pequeña escala, y consiste en ir probando uno a uno los diferentes módulos que constituyen una aplicación.

Las pruebas de integración y de aceptación son pruebas a mayor escala, que pueden llegar a dimensiones industriales cuando el número de módulos es muy elevado, o la funcionalidad que se espera del programa es muy compleja.

Las pruebas de integración se centran en probar la coherencia semántica entre los diferentes módulos, tanto de semántica estática (se importan los módulos adecuados; se llama correctamente a los procedimientos proporcionados por cada módulo), como de semántica dinámica (un módulo recibe de otro lo que esperaba).

Normalmente estas pruebas se van realizando por etapas, englobando progresivamente más y más módulos en cada prueba.

Las pruebas de integración se pueden empezar en cuanto tenemos unos pocos módulos, aunque no terminarán hasta disponer de la totalidad. En un

diseño descendente (top-dowm) se empieza a probar por los módulos más generales; mientras que en un diseño ascendente se empieza a probar por los módulos de base.

El planteamiento descendente tiene la ventaja de estar siempre pensado en términos de funcionalidad global, pero también tiene el inconveniente de que para cada prueba hay que “inventarse” algo sencillo (pero fiable) que simule el papel de los módulos inferiores, que aún no están disponibles.

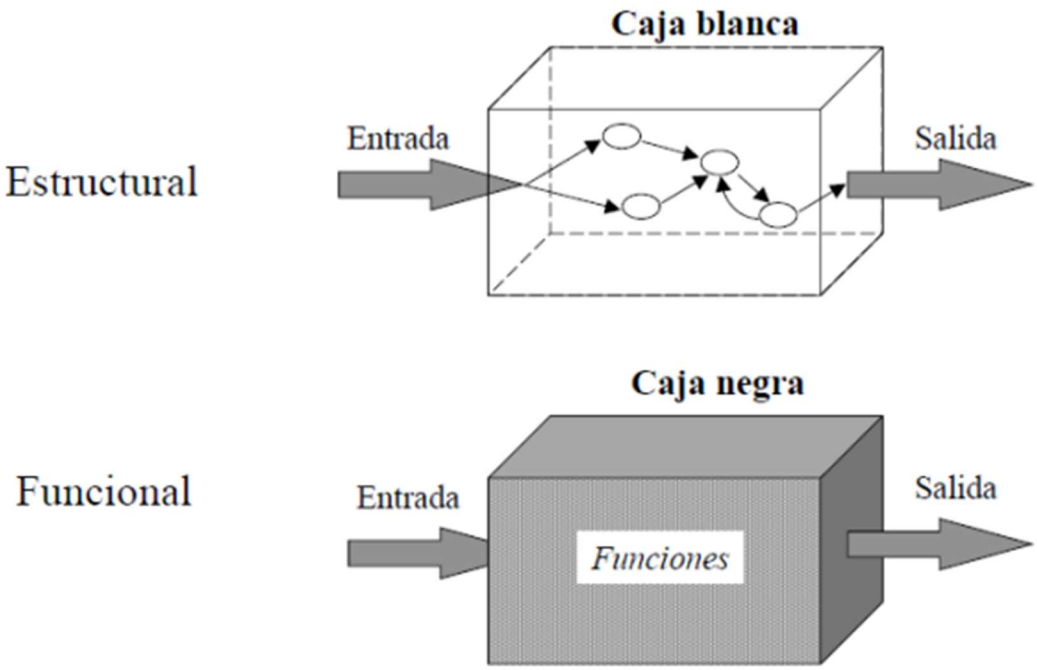
El planteamiento ascendente evita tener que escribirse módulos ficticios, pues vamos construyendo pirámides más y más altas con lo que vamos teniendo. Su desventaja es que se centra más en el desarrollo que en las expectativas finales del cliente.

Estas clasificaciones no son únicas. Por ejemplo, en sistemas con mucha interacción con el usuario es frecuente codificar sólo las partes de cada módulo que hacen falta para una cierta funcionalidad. Una vez probada, se añade otra funcionalidad y así hasta el final. Esto da lugar a un planteamiento más “vertical” de las pruebas. A veces se conoce como “codificación incremental”.

Por último, las pruebas de aceptación son las que se plantea el cliente final, que decide qué pruebas va a aplicarle al producto antes de darlo por bueno y pagarlo. De nuevo, el objetivo del que prueba es encontrar los fallos lo antes posible, en todo caso antes de pagarlo y antes de poner le programa en producción.

Existen tres enfoques principales para el diseño de casos:

1. El enfoque estructural o de caja blanca. Se centra en la estructura interna del programa (analiza los caminos de ejecución).
2. El enfoque funcional o de caja negra. Se centra en las funciones, entradas y salidas.
3. El enfoque aleatorio consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de pruebas.



3.1 PRUEBAS ESTRUCTURALES

También conocidas como pruebas de caja blanca o pruebas de caja transparente.

En estas pruebas estamos siempre observando el código, que las pruebas dedican a ejecutar con ánimo de “probarlo todo”. Esta noción de prueba total se formaliza en lo que se llama “cobertura” y no es sino una medida porcentual de ¿cuánto código hemos cubierto?

Hay diferentes posibilidades de definir la cobertura. Todas ellas intentan sobrevivir al hecho de que el número posible de ejecuciones de cualquier programa no trivial es (a todos los efectos prácticos) infinito. Pero si el 100% de cobertura es infinito, ningún conjunto real de pruebas pasaría de un infinitésimo de cobertura. Esto puede ser muy interesante para los matemáticos; pero no sirve para nada.

Cobertura de segmentos

A veces también denominada “cobertura de sentencias”. Por segmento se entiende una secuencia de sentencias sin punto de decisión. Como el ordenador está obligado a ejecutarlas una tras otras, es lo mismo decir que se

han ejecutado todas las sentencias o todos los segmentos.

El número de sentencias de un programa es finito. Basta coger el código fuente e ir contando. Se puede diseñar un plan de pruebas que vaya ejercitando más y más secuencias, hasta que hayamos pasado por todas, o por una inmensa mayoría.

En la práctica, el proceso de pruebas termina antes de llegar al 100%, pues puede ser excesivamente laborioso y costoso provocar el paso por todas y cada una de las sentencias.

A la hora de decidir el punto de corte antes de llegar al 100% de cobertura hay que ser precavido y tomar en consideración algo más que el índice conseguido. En efecto, ocurre con harta frecuencia que los programas contienen código muerto o inalcanzable. Puede ser que este trozo del programa, simplemente “sobre” y se pueda prescindir de él; pero a veces significa que una cierta funcionalidad, necesaria, es inalcanzable: esto es un error y hay que corregirlo.

Cobertura de ramas

La cobertura de segmentos es engañosa en presencia de segmentos opcionales. Por ejemplo:

```
IF Condición THEN EjecutaEsto; END;
```

Desde el punto de vista de cobertura de segmentos, hasta ejecutar una vez, con éxito en la condición, para cubrir todas las sentencias posibles. Sin embargo, desde el punto de vista de la lógica del programa, también debe ser importante el caso de que la condición falle (si no fuera, sobre el IF). Sin embargo, como en la rama ELSE no hay sentencias, con 0 ejecuciones tenemos el 100%.

Para afrontar estos casos, se plantea un refinamiento de la cobertura de segmentos consistente en recorrer todas las posibles salidas de los puntos de decisión. Para el ejemplo de arriba, para conseguir una cobertura de ramas del 100% hay que ejecutar al menos, 2 veces, una satisfaciendo la condición y otra no.

Nótese que, si lográramos una cobertura de ramas del 100%, esto llevaría implícita una cobertura del 100% de los segmentos, pues todo segmento está en alguna rama. Esto es cierto salvo en programas triviales que carecen de condiciones (a cambio, basta una sola prueba para cubrirlo desde todos los puntos de vista). El criterio también debe refinarse en lenguajes que admiten excepciones (como ADA). En estos casos, hay que añadir pruebas para provocar la ejecución de todas y cada una de las excepciones que pueden dispararse.

Cobertura de condición/decisión

La cobertura de ramas resulta a su vez engañosa cuando las expresiones booleanas que usamos para decidir por qué rama tirar son complejas. Por ejemplo:

```
IF Condicion1 OR Condicion2 THEN HazEsto; END;
```

Las condiciones 1 y 2 pueden tomar 2 valores cada una, dando lugar a 4 posibles combinaciones. No obstante, sólo hay dos posibles ramas y bastan 2 pruebas para cubrirlas. Pero con este criterio podemos estar cerrando los ojos a otras combinaciones de las condiciones.

Consideremos sobre el caso anterior las siguientes pruebas:

Prueba 1: Condicion1 = TRUE y Condicion2 = FALSE

Prueba 2: Condicion1 = FALSE y Condicion2 = TRUE

Prueba 3: Condicion1 = FALSE y Condicion2 = FALSE

Prueba 4: Condicion1 = TRUE y Condicion2 = TRUE

Bastan las pruebas 2 y 3 para tener cubiertas todas las ramas. Pero con ellos sólo hemos probado una posibilidad para la Condición 1.

Para afrontar esta problemática se define un criterio de cobertura de condición/decisión que trocea las expresiones booleanas complejas en sus componentes e intenta cubrir todos los posibles valores de cada uno de ellos.

Nótese que no basta con cubrir cada una de las condiciones componentes, sino que además hay que cuidar de sus posibles combinaciones de forma que se logre siempre probar todas y cada una de las ramas. Así, en el ejemplo anterior no basta con ejecutar las pruebas 1 y 2, pues aún cuando

cubrimos perfectamente cada posibilidad de cada condición por separado, lo que no hemos logrado es recorrer las dos posibles ramas de la decisión combinada. Para ello es necesario añadir la prueba 3.

El conjunto mínimo de pruebas para cubrir todos los aspectos es el formado por las pruebas 3 y 4. Aún así, nótese que no hemos probado todo lo posible. Por ejemplo, si en el programa nos colamos y ponemos AND donde queríamos poner OR (o viceversa), este conjunto de pruebas no lo detecta. Sólo queremos decir que la cobertura es un criterio útil y práctico; pero no es una prueba exhaustiva.

Cobertura de bucles

Los bucles no son más que segmentos controlados por decisiones. Así, la cobertura de ramas cubre plenamente la esencia de los bucles. Pero eso es simplemente la teoría, pues la práctica descubre que los bucles son una fuente inagotable de errores, todos triviales, algunos mortales. Un bucle se ejecuta en cierto número de veces, pero ese número de veces debe ser muy preciso, y lo más normal es que ejecutarlo una vez de menos o una vez de más tenga consecuencias indeseables. Y, sin embargo, es extremadamente fácil equivocarse y redactar un bucle que se ejecuta una vez de más o de menos.

Para un bucle del tipo WHILE hay que pasar 3 pruebas:

1. 0 ejecuciones
2. 1 ejecución
3. Más de 1 ejecución

Para un bucle de tipo REPEAT hay que pasar dos pruebas:

1. 1 ejecución
2. Más de una ejecución

Los bucles FOR, en cambio, son muy seguros, pues en su cabecera está definido el número de veces que se va a ejecutar. Ni una más, ni una menos, y el compilador se encarga de garantizarlo. Basta pues un ejecutarlos una vez.

No obstante, conviene no engañarse con los bucles FOR y examinar su contenido. Si dentro del bucle se altera la variable de control, o el valor de alguna variable que se utilice en el cálculo del incremento o del límite de iteración, entonces eso es un bucle FOR con trampa.

También tiene “trampa” si contiene sentencias del tipo EXIT (que algunos lenguajes denominan BREAK) o del tipo RETURN. Todas ellas provocan terminaciones anticipadas del bucle.

Estos últimos párrafos hay que precisarlos para cada lenguaje de programación.

Si el programa contine bucles LOOP, o simplemente bucles con trampa, la única cobertura aplicable es la de ramas. El riesgo de error es muy alto; pero no se conocen técnicas sistemáticas de abordarlo, salvo reescribir el código.

Y en la práctica ¿Cómo hago?

En la práctica de cada día, se suele procurar alcanzar la cobertura cercana al 100% de segmentos. Es muy recomendable (aunque cuesta más) conseguir una buena cobertura de ramas. En cambio, no suele hacer falta ir a por una cobertura de decisiones atomizadas.

¿Qué entenderemos por buena cobertura? Pues depende de lo crítico que sea el programa. Hay que valorar el riesgo (o coste) que implica un fallo si éste se descubre durante la aplicación del programa. Para la mayor parte del software que se produce en Occidente, el riesgo es simplemente de imagen (si un juego fallece a mitad, queda muy feo, pero no se muere nadie). En estas circunstancias, coberturas del 60-80% son admisibles.

La cobertura requerida suele ir creciendo con el ámbito previsto de distribución. Si un programa se distribuye y falla en algo grave puede ser necesario redistribuirlo de nuevo y urgentemente. Si hay millones de clientes dispersos por varios países, el coste puede ser brutal. En estos casos hay que exprimir la fase de pruebas para que se encuentre prácticamente todos los errores sin pasar nada por alto. Esto se traduce al final en buscar coberturas más altas.

Es aún más delicado cuando estamos en aplicaciones que involucran vidas humanas. Cuando un fallo se traduce en una muerte, la cobertura que se busca se acerca al 99% y además se presta atención a las decisiones atómicas.

La ejecución de pruebas de caja blanca puede llevarse a cabo con un depurador (que permite la ejecución paso a paso), un listado del módulo y un rotulador para ir marcando por dónde vamos pasando. Esta tarea es muy tediosa, pero suele ser automatizada.

Lograr una buena cobertura con pruebas de caja blanca es un objetivo deseable, pero no suficiente a todos los efectos. Un programa puede estar perfecto en todos sus términos y sin embargo no servir para la función que se pretende.

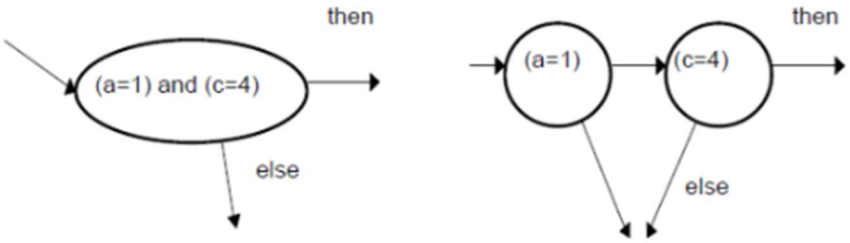
Por ejemplo, si escribimos una rutina para ordenar datos por orden ascendente, pero el cliente los necesita en orden descendente, no hay prueba de caja blanca capaz de detectar la desviación.

Criterios de cobertura lógica

Estos criterios van de menos rigurosos y más baratos a más rigurosos y por tanto, más caros:

- Cobertura de sentencias. Se trata de generar los casos de pruebas necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.
- Cobertura de decisiones. Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, un resultado falso.
- Cobertura de condiciones. Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez. (No incluye cobertura de condiciones)
- Criterio de decisión/condición. Consiste en exigir el criterio de cobertura de condiciones obligado a que se cumpla también el criterio de decisiones.

- Criterio de condición múltiple. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea, se puede considerar que cada decisión multicondicional se descompone en varias condiciones unicondicionales. Ejemplo:



- Criterio de cobertura de caminos. Se recorren todos los caminos (impracticable).

LA COMPLEJIDAD CICLOMÁTICA DE McCABEV (G) (COMPLEJIDAD CICOMÁTICA

La métrica de McCabe ha sido muy popular en el diseño de pruebas. Es un indicador del número de caminos independientes que existen en un grafo.

Se puede calcular de cualquiera de estas tres formas:

1. V (G) = a - n + 2, siendo a el número de arcos o aristas del grafo y n el número de nodos.

2. V (G) = r, siendo r el número de regiones cerradas del grafo.

3. V(G) = c + 1, siendo c el número de nodos de condición.

El criterio de la prueba de McCabe es: Elegir tantos casos de prueba como caminos independientes. La experiencia en este campo asegura que:

- V(G) marca el límite mínimo de casos de pruebas para un programa.
- Cuando V(G) > 10 la probabilidad de defectos en el módulo o programa crece mucho y por tanto, sería interesante dividir el módulo.

3.2 PRUEBAS FUNCIONALES

También conocida como **pruebas de caja opaca, pruebas de caja negra, pruebas de entrada/salida o pruebas inducidas por los datos.**

Las pruebas de caja negra se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en el que el módulo no se atiene a su especificación.

Por ello se denominan pruebas funcionales, y el probador se limita a suministrarle datos como entrada y estudiar la salida, sin preocuparse de lo que pueda estar haciendo el módulo por dentro.

Las pruebas de caja negra están especialmente indicadas en aquellos módulos que van a ser interfaz con el usuario (en sentido general: teclado, pantalla, ficheros, canales de comunicaciones, etc.) Este comentario no obste para que sean útiles en cualquier módulo del sistema.

Las pruebas de caja negra se apoyan en la especificación de requisitos del módulo. De hecho, se habla de “coberturas de especificación” para dar una medida del número de requisitos que se han probado. Es fácil obtener coberturas del 100% en módulos internos, aunque puede ser más laborioso en módulos con interfaz al exterior. En cualquier caso, es muy recomendable conseguir una alta cobertura en esta línea.

El problema con las pruebas de caja negra no suele estar en el número de funciones proporcionadas por el módulo (que siempre es un número muy limitado en diseños razonables); sino en los datos que se le pasan a estas funciones. El conjunto de datos posibles suele ser muy amplio (por ejemplo, un entero).

A la vista de los requisitos de un módulo, se sigue una técnica algebraica conocida como “clases de equivalencias”. En esta técnica trata cada parámetro como un modelo algebraico donde unos datos son equivalentes a otros. Si logramos partir un rango excesivamente amplio de posibles valores reales a un conjunto reducido de clases de equivalencia, entonces es suficiente probar un caso de cada clase, pues los demás datos de la misma clase son equivalentes.

El problema está pues en identificar clases de equivalencia, tarea para la que no existe una regla de aplicación universal; pero hay recetas para la mayor parte de los casos prácticos:

- Si un parámetro de entrada debe estar comprendido en un cierto rango, aparecen tres clases de equivalencia: por debajo, en y por encima del rango.
- Si una entrada requiere un valor concreto, aparecen 3 clases de equivalencia: por debajo, en y por encima del rango.
- Si una entrada requiere un valor de entre los de un conjunto, aparecen 2 clases de equivalencia: en el conjunto o fuera de él.
- Si una entrada es booleana, hay 2 clases: si o no.
- Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

Durante la lectura de los requisitos del sistema, nos encontraremos con una serie de valores singulares, que marcan diferencias de comportamiento. Estos valores son claros candidatos a marcar clases de equivalencia: por debajo y por arriba.

Una vez identificadas las clases de equivalencias significativas en nuestro módulo, se procede a coger un valor de cada clase, que no esté justamente al límite de la clase. Este valor aleatorio, hará las veces de cualquier valor normal que se le pueda pasar en la ejecución real.

La experiencia muestra que un buen número de errores aparecen en torno a los puntos de cambio de clase de equivalencia. Hay una serie de valores denominados “fronteras” (o valores límites) que conviene probar, además de los elegidos en el párrafo anterior. Usualmente se necesitan 2 valores por frontera, uno justo abajo y otro justo encima.

Limitaciones

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable; pero no suficiente a todos los efectos. Un programa puede pasar con holgura millones de pruebas y sin embargo tener defectos internos que surgen en el momento más inoportuno.

Las pruebas de caja negra nos convencen de que un programa hace lo que queremos; pero no de que además haga otras cosas menos aceptables.

Se centra en las funciones de entrada y salida. Es impracticable probar el software para todas las posibilidades. De nuevo hay que tener criterios para elegir buenos casos de prueba.

Un caso de prueba funcional es bien elegido si se cumple que:

- Reduce el número de otros casos necesarios para que la prueba sea razonable. Esto implica que el caso ejecute el máximo número de posibilidades de entrada diferentes para así reducir el total de casos.
- Cubre un conjunto extenso de otros casos posibles, es decir, no indica algo acerca de la ausencia o la presencia de defectos en el conjunto específico de entradas que prueba, así como de otros conjuntos similares.

4 PRUEBAS UNITARIAS

En apartados anteriores hemos hablado de las pruebas de caja blanca, las cuales son las primeras pruebas que se deben realizar, las pruebas unitarias son pruebas de caja blanca. Anteriormente hemos hablado de las diferentes técnicas y metodologías para diseñar las pruebas de caja blanca, ahora vamos a ver cómo realizarlas una vez diseñadas.

Sin duda, las pruebas unitarias son una labor fundamenta en el trabajo de todo desarrollador, no obstante, suelen ser creadas y ejecutadas por personal especializado en pruebas de software.

Las pruebas unitarias son pruebas individuales para un método o clase, realizadas de manera sistemática a modo de batería de pruebas donde conocemos los datos de entrada y sabemos cuál sería el resultado esperado.

Hoy en día existen una serie de herramientas que facilitan esta tarea: desde la programación de pruebas automáticas que se crean cada vez que se implementa una nueva funcionalidad o método, o las más usuales, que nos permiten ejecutar las baterías de pruebas de una manera ordenada visualizando todos los resultados de manera directa y eficaz.

Cabe resaltar que este tipo de pruebas son tremendamente útiles, prácticamente imprescindibles en el desarrollo colaborativo.

4.1 Metodologías

Las pruebas deberían implementarse de manera sistemática con cada nueva funcionalidad que se añada, teniendo de este modo las pruebas actualizadas en cada momento. Es importante que las pruebas se realicen y se ejecuten de manera incremental de este modo, aunque hayamos probado una funcionalidad anterior en varias ocasiones con los resultados esperados, comprobaríamos en todas las pruebas que dicho funcionamiento no se ha visto alterado.

Es recomendable que, antes de implementar una nueva funcionalidad, primero pensemos cómo deberíamos probarla para revisar que se ejecuta correctamente. De este modo, nos permite llevar un desarrollo en donde

tenemos las ideas muy claras y podemos crear las pruebas inmediatamente después de codificar la funcionalidad sin que nuestro rendimiento se vea afectado.

4.2 Junit

Junit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas en aplicaciones Java. Se trata de una herramienta ampliamente extendida en el desarrollo de aplicaciones Java, y que está en continua actualización, sumando nuevas funcionalidades cada poco tiempo. Actualmente se encuentra en la versión Junit 5, y su web oficial es https://junit.org/junit5/

Se trata de un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado, si la clase cumple con la especificación, entonces Junit devolverá que el método de la clase pasó exitosamente la prueba. En caso de que el valor esperado sea diferente al que retornó el método durante la ejecución, Junit devolverá un fallo en el método correspondiente.

Junit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionamiento después de la nueva modificación.

El uso de Junit lo veremos en clase con la practica 2.