Pruebas de Programas

‽ ***José A. Mañas***
‽ ***Gabriel Huecas***
‽ ***Tomás Robles***

<jmanas@dit.upm.es>
<gabriel@dit.upm.es>
<robles@dit.upm.es>

28 de Febrero de 2002

ÍNDICE

INTRODUCCIÓN 4

¿QUÉ ES PROBAR? 4
LA PRUEBA EXHAUSTIVA ES IMPOSIBLE 4
ORGANIZACIÓN 5

PRUEBA DE UNIDADES 6

CAJA BLANCA 7
Cobertura de trazas 7
Cobertura de segmentos 8
Cobertura de ramas 8
Cobertura de decisiones 8
Cobertura de bucles 9
Y en la práctica ¿qué hago? 10
Limitaciones 11
CAJA NEGRA 11
Limitaciones 12
PRUEBAS DE INTEGRACIÓN 13
PRUEBAS DE ACEPTACIÓN 13
OTROS TIPOS DE PRUEBAS 14
Recorridos (walkthroughs) 14
Aleatorias (random testing) 14
Solidez (robustness testing) 14
Aguante (stress testing) 15
Prestaciones (performance testing) 15
Conformidad u Homologación (conformance testing) 15
Interoperabilidad (interoperability tesing) 15
Regresión (regression testing) 15
Mutación (mutation testing) 16

DEPURACIÓN (DEBUGGING) 16

PLAN DE PRUEBAS 16

GENERACIÓN DE CASOS DE PRUEBA 17
Fases de una prueba 17
Propósitos de Prueba 17
Temporizadores 18
Selección de Datos de Prueba 18
Clases de Pruebas 18

EJECUCIÓN DE LAS PRUEBAS 19

Imposición versus Observación 19
Seguimiento Sobre la Especificación de Referencia 19
Bloqueo 20
Pruebas correctas 20
ASIGNACIÓN DE VEREDICTOS 21

PRUEBA DE PROGRAMAS ORIENTADOS A OBJETOS 21

LA NATURALEZA DE LA POO 22
PROBLEMAS ESPECÍFICOS DE LA POO 22
La clase como unidad de Pruebas *22*
Encapsulado *23*
Herencia *23*
Polimorfismo *23*

ASPECTOS SICOLÓGICOS Y ORGANIZACIÓN DEL TRABAJO 24

CONCLUSIONES 24

BIBLIOGRAFÍA 25

GLOSARIO 26

CASO PRÁCTICO 26

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 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 puede fácilmente superar el 80%.

Pese a su enorme impacto en el coste de 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.

¿Qué es probar?

Mientras que para un matemático probar es poco mas o menos demostrar la corrección de un programa, para un programador es básicamente convencerse de que el programa va bien, funciona correctamente, y tendrá éxito y aceptación cuando lo entregue a sus usuarios finales.

El IEEE se atreve con una definición:

> Es el proceso de ejercitar o evaluar un sistema, manual o automáticamente, con el ánimo de verificar que satisface los requisitos especificados, o identificar discrepancias entre los resultados esperados y los que el programa devuelve.

La práctica nos convence, en cambio, de que hay que usar planteamientos más duros, del tipo:

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

Por poner un ejemplo duro, probar un programa es equivalente a la actividad de ciertos profesores para los que examinar a un alumno consiste en poner en evidencia todo lo que no sabe. Esto es penoso cuando se aplica a personas; pero es exactamente lo que hay que hacerle a los programas.

La Prueba Exhaustiva es Imposible

La prueba ideal de un sistema sería ponerlo en todas las situaciones posibles, comprobar que se comporta bien en todas y cada una de ellas, y así estar seguros de su respuesta ante cualquier caso que se le presente en 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 código, 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 bucles, en los que es fácil introducir condiciones para un funcionamiento sin fin. Aún en el irreal 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 someterle a todas las posible variaciones de los datos de entrada, tanto si son válidos como si no lo son. Imagínese 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 conseguir un nivel de confianza alto en lo que estamos entregando al usuario. Nótese que todo es muy relativo y resbaladizo en este área.

Organización

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

Fases de prueba:

- UNIDADES

  Planteamientos:

  - CAJA BLANCA

    Cobertura:

    - de segmentos
    - de decisiones
    - 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 puede 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 estática (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-down) se

empieza a probar por los módulos más generales; mientras que en un diseño descendente se empieza a probar por los módulos de base.

El planteamiento descendente tiene la ventaja de estar pensado en términos de la 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 las únicas posibles. 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".

Aunque hay casos para todos los gustos, parece que lo más habitual es un plan ascendente, por el ahorro de codificación que supone.

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.

Prueba de Unidades

¿Cómo se prueban módulos sueltos?

Normalmente cabe distinguir entre una fase informal y una fase sistemática. La fase informal la lleva a cabo el propio codificador en su despacho, y consiste en ir ejecutando el código para convencerse de que "básicamente, funciona". Esta fase suele consistir en pequeños ejemplos que se intentan ejecutar. Si el módulo falla, se suele utilizar un depurador para observar la evolución dinámica del sistema, localizar el fallo, y repararlo.

En lenguajes antiguos, poco rigurosos en la sintaxis y/o en la semántica de los programas, esta fase informal llega a ser muy dura, laboriosa, y susceptible de dejar pasar grandes errores sin que se note. En lenguajes modernos, con reglas estrictas, hay herramientas que permiten análisis exhaustivos de los aspectos estáticos de la semántica de los programas: tipado de las variables, ámbitos de visibilidad, parámetros de llamada a procedimientos, etc.

Hay así mismo herramientas más sofisticadas capaces de emitir "opiniones" sobre un programa y alertar de construcciones arriesgadas, de expresiones muy complicadas (que se prestan a equivocaciones), etc. A veces pueden prevenir sobre variables que pueden usarse antes de tomar algún valor (no inicializadas), variables que se cargan pero luego no se usan, y otras posibilidades que, sin ser necesariamente errores en sí mismas, si suelen apuntar a errores de verdad.

Más adelante, cuando el módulo parece presentable, se entra en una fase de prueba sistemática. En esta etapa se empieza a buscar fallos siguiendo algún criterio para que "no se escape nada". Los criterios más habituales son los denominados de caja negra y de caja blanca.

Se dice que una prueba es de caja negra cuando prescinde de los detalles del código y se limita a lo que se ve desde el exterior. Intenta descubrir casos y circunstancias en los que el módulo no hace lo que se espera de él.

Por oposición al término “caja negra” se suele denominar “caja blanca” al caso contrario, es decir, cuando lo que se mira con lupa es el código que está ahí escrito y se intenta que falle. Quizás sea más propio la denominación de “pruebas de caja transparente”.

Caja blanca

Sinónimos:

?? pruebas estructurales

?? pruebas de caja transparente

En estas pruebas estamos siempre observando el código, que las pruebas se 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. Es deseable fijar un límite deseable y alcanzable que nos determine el 100%. Si no se pasa ninguna prueba, no hemos “cubierto” ninguna parte del código y diremos que tenemos un 0% de cobertura. A medida que se ejecuten pruebas sobre el sistema la cobertura ira aumentando de forma monótona no decreciente, hasta alcanzar dicho 100%. En todo caso, por arbitrario que sea el límite, se debe cumplir que 0 <= C <= 100 para cualquier valor de cobertura C expresado porcentualmente. Evidentemente, el límite del 100% nos indica que debemos de parar de ejecutar pruebas.

Los criterios para definir la cobertura son, como hemos dicho, artificiales. No obstante, deben reflejar alguna característica o o propiedad que ayuden a interpretar los valores obtenidos en la asignación de cobertura a la ejecución de una prueba. Por ello se pide que:

? Sea fácil de definir

? Sea fácil de medir

? Sea fácil de entender

Puesto que se intentan diseñar pruebas para ejecutar sobre programas, usaremos los elementos que conforman dichos programas para definir los criterios de cobertura.

Cobertura de trazas

En algunos programas sencillos el número posible de trazas de ejecución es muy limitado. En estos casos puede ser útil utilizar el número de trazas observadas / número de trazas posibles como índice de cobertura. Según se va consiguiendo observar más y más trazas, se va acercando el proceso a una cobertura del 100%.

En la práctica, el proceso de pruebas termina antes, pues puede ser excesivamente laborioso y costoso provocar todas y cada una de las trazas posibles.

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 contengan 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 es inalcanzable.

Cobertura de segmentos

A veces también denominada “cobertura de sentencias”. Por segmento se entiende una secuencia de sentencias sin puntos de decisión. Como el ordenador está obligado a ejecutarlas una tras otra, 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 sentencias, 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.

De nuevo tenemos que la existencia de código muerto implique la imposibilidad de llegar al 100% de cobertura.

Un valor del 100% en cobertura de segmentos implica un valor del 100% en cobertura de trazas, puesto que toda traza está incluida en algún segmento.

Cobertura de ramas

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

```
IF TalCondicion THEN EjecutaEsto; END;
```

Desde el punto de vista de cobertura de segmentos, basta 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 lo fuera, sobra 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.

Estos criterios se extienden a las construcciones que suponen elegir 1 de entre varias ramas. Por ejemplo, el CASE. En ete caso se deben diseñar pruebas que ejerciten todas las ramas del CASE, incluyendo el ELSE.

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.

Cobertura de decisiones

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:

1. Prueba 1: Condicion1 = TRUE y Condicion2 = FALSE

2. Prueba 2: Condicion1 = FALSE y Condicion2 = TRUE
3. Prueba 3: Condicion1 = FALSE y Condicion2 = FALSE
4. 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 Condicion1.

Para afrontar esta problemática se define un criterio de cobertura de 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, si no 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 prueba exhaustiva.

Una dificultad añadida surge cuando las condiciones no son mutuamente excluyentes. Por ejemplo, puede ocurrir que Condicion1 implique Condicion2. Esto se da cuando si Condicion1 es TRUE, Condicion2 también lo es, pero si Condicion1 es FALSE, Condicion2 puede ser TRUE o FALSE.

Aparte de las razones apuntadas en los casos anteriores que imposibilitan alcanzar el 100% de cobertura, en este caso puede haber condiciones que nunca / siempre se evalúen a TRUE o FALSE. Estos casos suelen indicar la presencia de algún error: o bien la sentencia IF es totalmente innecesaria o bien existe un error en la condición. Correspondientemente, en el caso del CASE su expresión asociada puede no evaluarse a ciertos valores.

Cobertura de bucles

Los bucles no son mas 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 un cierto número de veces; pero ese número de veces debe ser muy preciso, muy 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 1 vez de más o de menos.

Para un bucle de tipo WHILE hay que pasar 3 pruebas

?? 0 ejecuciones

?? 1 ejecución

?? más de 1 ejecución

Para un bucle de tipo REPEAT hay que pasar 2 pruebas

?? 1 ejecución

?? más de 1 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 con ejecutarlos 1 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. Lo peor son aquellos lenguajes que permiten el uso de sentencias GOTO. Tampoco conviene confiarse de lo que prometen lenguajes como MODULA-2, que se supone que prohíben ciertas construcciones arriesgadas. Los compiladores reales suelen ser mas tolerantes que lo que anuncian los libros.

Si el programa contiene 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 rescribir el código.

Y en la práctica ¿qué hago?

En la práctica de cada día, se suele considerar casi imprescindible una buena cobertura de segmentos. Es muy recomendable (y cuesta poco más) conseguir una buena cobertura de ramas. En cambio, no suele hacer falta ir a por una cobertura de decisiones atomizadas.

¿Qué es una buena cobertura?

Pues depende de lo crítico que sea el programa. Hay que valorar el riesgo (o coste) que implica un fallo si este 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 70-80% 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 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 entramos en aplicaciones que involucran vidas humanas (aplicaciones sanitarias, centrales nucleares, etc). 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.

También se suele perseguir coberturas muy elevadas (por encima del 90%) en las aplicaciones militares. Esto se debe a que normalmente van a ser utilizadas en condiciones muy adversas donde el tiempo es inestimable. Si un programa fallece, puede no haber una segunda oportunidad de arrancarlo de nuevo.

La ejecución de pruebas de caja blanca puede llevarse a cabo con un depurador (que permite le ejecución paso a paso), un listado del módulo y un rotulador para ir marcando por dónde vamos pasando. Esta tarea es muy tediosa, pero puede ser automatizada. Hay compiladores que a la hora de generar código máquina dejan incrustado en el código suficiente código como para poder dejar un fichero (tras la ejecución) con el número de veces que se ha ejecutado cada sentencia, rama, bucle, etc.

Limitaciones

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 a la función que se pretende.

Por ejemplo, un Rolls-Royce es un coche que sin duda pasaría las pruebas más exigentes sobre los últimos detalles de su mecánica o su carrocería. Sin embargo, si el cliente desea un todo-terreno, difícilmente va a comprárselo.

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

Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace; pero no de que haga lo que necesitamos.

Caja negra

Sinónimos:

?? pruebas de caja opaca

?? pruebas funcionales

?? pruebas de entrada/salida

?? pruebas inducidas por los datos

Las pruebas de caja negra se centran en lo que se espera de un módulo, es decir que intenta encontrar casos en 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 obsta 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 “cobertura 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 son 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 equivalencia". 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 3 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.

Ejemplo: utilizamos un entero para identificar el día del mes. Los valores posibles están en el rango [1..31]. Así, hay 3 clases:

- números menores que 1
- números entre 1 y 31
- números mayores que 31

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 abajo y por arriba.

Una vez identificadas las clases de equivalencia significativas en nuestro módulo, se procede a coger un valor de cada clase, que no está 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 de nominados "frontera" 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 para convencernos de que es lo que necesitamos, y sin embargo tener defectos internos que surgen en el momento más inoportuno.

Por ejemplo, un PC que contenga el virus Viernes-13 puede estar pasando pruebas de caja negra durante años y años. Sólo falla si es viernes y es día 13; pero ¿a quién se le iba a ocurrir hacer esa prueba?

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

Pruebas de Integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema, involucran a un número creciente de módulos y terminan probando el sistema como conjunto.

Estas pruebas se pueden plantear desde un punto de vista estructural o funcional.

Las pruebas estructurales de integración son similares a las pruebas de caja blanca; pero trabajan a un nivel conceptual superior. En lugar de referirnos a sentencias del lenguaje, nos referiremos a llamadas entre módulos. Se trata pues de identificar todos los posibles esquemas de llamadas y ejercitarlos para lograr una buena cobertura de segmentos o de ramas.

Las pruebas funcionales de integración son similares a las pruebas de caja negra. Aquí trataremos de encontrar fallos en la respuesta de un módulo cuando su operación depende de los servicios prestados por otro(s) módulo(s). Según nos vamos acercando al sistema total, estas pruebas se van basando más y más en la especificación de requisitos del usuario.

Las pruebas finales de integración cubren todo el sistema y pretenden cubrir plenamente la especificación de requisitos del usuario. Además, a estas alturas ya suele estar disponible el manual de usuario, que también se utiliza para realizar pruebas hasta lograr una cobertura aceptable.

En todas estas pruebas funcionales se siguen utilizando las técnicas de partición en clases de equivalencia y análisis de casos límite (fronteras).

Pruebas de Aceptación

Estas pruebas las realiza el cliente. Son básicamente pruebas funcionales, sobre el sistema completo, y buscan una cobertura de la especificación de requisitos y del manual del usuario. Estas pruebas no se realizan durante el desarrollo, pues sería impresentable de cara al cliente; sino una vez pasadas todas las pruebas de integración por parte del desarrollador.

La experiencia muestra que aún después del más cuidadoso proceso de pruebas por parte del desarrollador, quedan una serie de errores que sólo aparecen cuando el cliente se pone a usarlo. Los desarrolladores se suelen llevar las manos a la cabeza:

“Pero, ¿a quién se le ocurre usar así el sistema?”

Sea como sea, el cliente siempre tiene razón. Decir que los requisitos no estaban claros, o que el manual es ambiguo puede salvar la cara; pero ciertamente no deja satisfecho al cliente.

Por estas razones, muchos desarrolladores ejercitan unas técnicas denominadas “pruebas alfa” y “pruebas beta”. Las pruebas alfa consisten en invitar al cliente a que venga al entorno de desarrollo a probar el sistema. Se trabaja en un entorno controlado y el cliente siempre tiene un experto a mano para ayudarle a usar el sistema y para analizar los resultados.

Las pruebas beta vienen después de las pruebas alfa, y se desarrollan en el entorno del cliente, un entorno que está fuera de control. Aquí el cliente se queda a solas con el producto y trata de encontrarle fallos (reales o imaginarios) de los que informa al desarrollador.

Las pruebas alfa y beta son habituales en productos que se van a vender a muchos clientes. Algunos de los potenciales compradores se prestan a estas pruebas bien por ir entrenando a su personal con tiempo, bien a cambio de alguna ventaja económica (mejor precio sobre el producto final, derecho a mantenimiento gratuito, a nuevas versiones, etc.). La experiencia muestra que estas prácticas son muy eficaces.

Otros tipos de pruebas

Recorridos (walkthroughs)

Quizás es una técnica más aplicada en control de calidad que en pruebas. Consiste en sentar alrededor de una mesa a los desarrolladores y a una serie de críticos, bajo las órdenes de un moderador que impida un recalentamiento de los ánimos. El método consiste en que los revisores se leen el programa línea a línea y piden explicaciones de todo lo que no está meridianamente claro. Puede que simplemente falte un comentario explicativo, o que detecten un error auténtico o que simplemente el código sea tan complejo de entender/explicar que más vale que se rehaga de forma más simple. Para un sistema complejo pueden hacer falta muchas sesiones.

Esta técnica es muy eficaz localizando errores de naturaleza local; pero falla estrepitosamente cuando el error deriva de la interacción entre dos partes alejadas del programa. Nótese que no se está ejecutando el programa, sólo mirándolo con lupa, y de esta forma sólo se ve en cada instante un trocito del listado.

Aleatorias (random testing)

Ciertos autores consideran injustificada una aproximación sistemática a las pruebas. Alegan que la probabilidad de descubrir un error es prácticamente la misma si se hacen una serie de pruebas aleatoriamente elegidas, que si se hacen siguiendo las instrucciones dictadas por criterios de cobertura (caja negra o blanca).

Como esto es muy cierto, probablemente sea muy razonable comenzar la fase de pruebas con una serie de casos elegidos al azar. Esto pondrá de manifiesto los errores más patentes. No obstante, pueden permanecer ocultos errores más sibilinos que sólo se muestran ante entradas muy precisas.

Si el programa es poco crítico (una aplicación personal, un juego, ...) puede que esto sea suficiente. Pero si se trata de una aplicación militar o con riesgo para vidas humanas, es de todo punto insuficiente. Incluso en programas de riesgo nulo, puede ser insuficiente si se prevé una distribución muy amplia (simplemente por el número de copias, disquetes, manuales, etc. que habrá que revisar cuando se descubra un error más tarde).

Solidez (robustness testing)

Se prueba la capacidad del sistema de salir de situaciones embarazosas provocadas por errores en el suministro de datos. Estas pruebas son importantes en sistemas con una interfaz al exterior, en particular cuando la interfaz es humana.

Por ejemplo, en un sistema que admite una serie de órdenes (commands) se deben probar los siguientes extremos:

- órdenes correctas, todas y cada una
- órdenes con defectos de sintaxis, tanto pequeñas desviaciones como errores de bulto
- órdenes correctas, pero en orden incorrecto, o fuera de lugar
- la orden nula (línea vacía, una o más)
- órdenes correctas, pero con datos de más
- provocar una interrupción (BREAK, ^C, o lo que corresponda al sistema soporte) justo después de introducir una orden.

?? órdenes con delimitadores inapropiados (comas, puntos, ...)

?? órdenes con delimitadores incongruentes consigo mismos (por ejemplo, esto]

Aguante (stress testing)

En ciertos sistemas es conveniente saber hasta dónde aguantan, bien por razones internas (¿hasta cuantos datos podrá procesar?), bien externas (¿es capaz de trabajar con un disco al 90%?, ¿aguanta una carga de la CPU del 90?, etc.)

Prestaciones (performance testing)

A veces es importante el tiempo de respuesta, u otros parámetros de gasto. Típicamente nos puede preocupar cuánto tiempo le lleva al sistema procesar tantos datos, o cuánta memoria consume, o cuánto espacio en disco utiliza, o cuántos datos transfiere por un canal de comunicaciones, o ... Para todos estos parámetros suele ser importante conocer cómo evolucionan al variar la dimensión del problema (por ejemplo, al duplicarse el volumen de datos de entrada).

Conformidad u Homologación (conformance testing)

En programas de comunicaciones es muy frecuente que, además de los requisitos específicos del programa que estamos construyendo, aparezca alguna norma más amplia a la que el programa deba atenerse. Es frecuente que organismos internacionales como ISO y el CCITT elaboren especificaciones de referencia a las que los diversos fabricantes deben atenerse para que sus ordenadores sean capaces de entenderse entre sí.

Las pruebas, de caja negra, que se le pasan a un producto para detectar discrepancias respecto a una norma de las descritas en el párrafo anterior se denominan de conformidad u homologación. Suelen realizarse en un centro especialmente acreditado al efecto y, si se pasan satisfactoriamente, el producto recibe un sello oficial que dice: "homologado".

Interoperabilidad (interoperability tesing)

En el mismo escenario del punto anterior, programas de comunicaciones que deben permitir que dos ordenadores se entiendan, aparte de las pruebas de conformidad se suelen correr una serie de pruebas, también de caja negra, que involucran 2 o más productos, y buscan problemas de comunicación entre ellos.

Regresión (regression testing)

Todos los sistemas sufren una evolución a lo largo de su vida activa. En cada nueva versión se supone que o bien se corrigen defectos, o se añaden nuevas funciones, o ambas cosas. En cualquier caso, una nueva versión exige una nueva pasada por las pruebas. Si estas se han sistematizado en una fase anterior, ahora pueden volver a pasarse automáticamente, simplemente para comprobar que las modificaciones no provocan errores donde antes no los había.

El mínimo necesario para usar unas pruebas en una futura revisión del programa es una documentación muy clara.

Las pruebas de regresión son particularmente espectaculares cuando se trata de probar la interacción con un agente externo. Existen empresas que viven de comercializar productos que "graban" la ejecución de una prueba con operadores humanos para luego repetirla cuantas veces haga falta "reproduciendo la grabación". Y, obviamente, deben monitorizar la respuesta del sistema en ambos casos, compararla, y avisar de cualquier discrepancia significativa.

Mutación (mutation testing)

Es una técnica curiosa consistente en alterar ligeramente el sistema bajo pruebas (introduciendo errores) para averiguar si nuestra batería de pruebas es capaz de detectarlo. Si no, más vale introducir nuevas pruebas. Todo esto es muy laborioso y francamente artesano.

Depuración (debugging)

Casi todos los compiladores suelen llevar asociada la posibilidad de ejecutar un programa paso a paso, permitiéndo al operador conocer dónde está en cada momento, y cuánto valen las variables.

Los depuradores pueden usarse para realizar inspecciones rigurosas sobre el comportamiento dinámico de los programas. La práctica demuestra, no obstante, que su uso es tedioso y que sólo son eficaces si se persigue un objetivo muy claro. El objetivo habitual es utilizarlo como consecuencia de la detección de un error. Si el programa se comporta mal en un cierto punto, hay que averiguar la causa precisa para poder repararlo. La causa a veces es inmediata (por ejemplo, un operador booleano equivocado); pero a veces depende del valor concreto de los datos en un cierto punto y hay que buscar la causa en otra zona del programa.

En general es mala idea "correr al depurador", tanto por el tiempo que se pierde buceando sin una meta clara, como por el riesgo de corregir defectos intermedios sin llegar a la raíz del problema. Antes de entrar en el depurador hay que delimitar el error y sus posibles causas. Ante una prueba que falla, hay que identificar el dominio del fallo, averiguar las características de los datos que provoca el fallo (y comprobar experimentalmente que todos los datos con esas características provocan ese fallo, y los que no las tienen no lo provocan).

El depurador es el último paso para convencernos de nuestro análisis y afrontar la reparación con conocimiento de causa.

Plan de Pruebas

Un plan de pruebas está constituido por un conjunto de pruebas. Cada prueba debe:

- dejar claro qué tipo de propiedades se quieren probar (corrección, robustez, fiabilidad, amigabilidad, ...)
- dejar claro cómo se mide el resultado
- especificar en qué consiste la prueba (hasta el último detalle de cómo se ejecuta)
- definir cual es el resultado que se espera (identificación, tolerancia, ...)
- ¿Cómo se decide que el resultado es acorde con lo esperado?

Las pruebas angelicales carecen de utilidad, tanto si no se sabe exactamente lo que se quiere probar, o si no está claro cómo se prueba, o si el análisis del resultado se hace "a ojo".

Estas mismas ideas se suelen agrupar diciendo que un caso de prueba consta de 3 bloques de información:

1) El propósito de la prueba
2) Los pasos de ejecución de la prueba
3) El resultado que se espera

Y todos y cada uno de esos puntos debe quedar perfectamente documentado. Las pruebas de usar y tirar más vale que se tiren directamente, aún antes de usarlas.

Cubrir estos puntos es muy laborioso y, con frecuencia, tedioso, lo que hace muy desagradable (o al menos muy aburrida) la fase de pruebas. Es mucho más divertido codificar que probar. Tremendo error en el que, no obstante, se cae casi siempre.

Respecto al orden de pruebas, una práctica frecuente es la siguiente:

1. Pasar pruebas de caja negra analizando valores límite. Recuerde que hay que analizar condiciones límite de entrada y de salida.
2. Identificar clases de equivalencia de datos (entrada y salida) y añadir más pruebas de caja negra para contemplar valores normales (en las clases de equivalencia en que estos sean diferentes de los valores límite; es decir, en rangos amplios de valores).
3. Añadir pruebas basadas en "presunción de error". A partir de la experiencia y el sentido común, se aventuran situaciones que parecen proclives a padecer defectos, y se buscan errores en esos puntos. Son pruebas del tipo "!Me lo temía!".
4. Medir la cobertura de caja blanca que se ha logrado con las fases previas y añadir más pruebas de caja blanca hasta lograr la cobertura deseada. Normalmente se busca una buena cobertura de ramas (revise los comentarios expuestos al hablar de caja blanca).

Generación de Casos de Prueba

La obtención de la batería de pruebas es una fase previa orientada a organizar técnica y administrativamente la realización de las pruebas.

Fases de una prueba

Los casos de prueba constan de un **preámbulo**, un **cuerpo** y un **postámbulo** [ISO:9646]. El preámbulo es una fase preliminar en la que la implementación se lleva al estado en el que se quiere realizar la prueba. En pruebas muy sencillas, este preámbulo es nulo. El cuerpo es el objeto en sí de la prueba. Los preámbulos de pruebas complejas son el cuerpo de pruebas preliminares. Por último, el postámbulo es el conjunto de acciones que nos permiten llevar la implementación a un estado reconocible tras una prueba. A veces es tan simple como apagar y volver a empezar.

El caso de prueba debe guardar información de aquellos posibles caminos que el sistema puede abordar indeterminísticamente, pues impedirán conseguir el objetivo de la prueba, pero no podrán ser considerados como errores.

Propósitos de Prueba

Las pruebas se organizan por propósitos. Estos consisten en identificar un estado inicial, al que llega gracias al preámbulo, y un objetivo que es el cuerpo de la prueba.

Un propósito de prueba es una descripción en prosa de un objetivo de prueba definido con precisión, centrado en un único requisito, de acuerdo con la especificación del sistema (adaptado de [ISO:9646]).

Existen dos aspectos importantes en esta definición. Por un lado, cada prueba *debe probar un único requisito o propósito*; si es cierto que, a veces, diferentes propósitos dan lugar a pruebas iguales o muy parecidas. Pero el objetivo *es distinto*. Por otro lado, ese requisito debe extraerse de la denominada especificación del producto, que puede estar normalizada.

Temporizadores

Hay que prever que la implementación falle y que durante las pruebas algún evento previsto en la prueba no se lleve a cabo. La decisión de que un evento no ocurre debe tomarse en base al vencimiento de un temporizador:

"si no ocurre en T segundos, considérese que no va a ocurrir nunca".

En definitiva, contestamos a "¿cuánto tiempo espero a que algo ocurra?". Esta información sobre tiempos de espera va estrechamente asociada a los propósitos de prueba, pudiéndose expresar bien en términos globales (idéntico criterio para todos los eventos) o particulares (usando temporizadores específicos para cada caso de prueba o, incluso, para algunos eventos).

Selección de Datos de Prueba

La selección de los datos de prueba es uno de los puntos más difíciles en el diseño de las pruebas. No basta con disponer de un propósito y saber con detalle las trazas que lo comprueban, además hay que proporcionar datos concretos para llevar a cabo la ejecución. Es usualmente imposible probar todos los casos con todos los datos posibles, pues éstos suelen ser infinitos, o tan numerosos que hacen inviable en la práctica su ejercicio exhaustivo. Hay que limitarse a un subconjunto, que deberá ser mínimo, pero maximizar la probabilidad de encontrar errores en el producto. Para ello hay que partir el dominio de datos de entrada en un conjunto finito de *clases de equivalencia* de forma que pueda razonablemente suponerse (aunque nunca se sepa con certeza absoluta) que la prueba con un valor cualquiera de la clase sea equivalente a la prueba con cualquier otro valor de la misma. La identificación de estas clases de equivalencia es un proceso heurístico en el que se toma la especificación del sistema y se recorren las condiciones de la misma. En base a su inspección, los datos quedan clasificados en clases según se cumpla o incumpla cada combinación de condiciones sobre ellos.

Como ya se ha comentado, una vez identificada una clase de equivalencia, se toma un elemento medio representativo de cada una de ellas, así como elementos límites, es decir valores que se hallan inmediatamente arriba o debajo de los márgenes de la clase. De esta forma, cada clase de equivalencia y cada frontera entre clases de equivalencia es probada.

Clases de Pruebas

En la fase de generación de la batería de pruebas se hace un análisis exhaustivo buscando secuencias de acciones que recorran los objetivos de pruebas. De este análisis resultan los casos de prueba clasificados en tres clases:

?? **obligatorios**: Una cierta prueba es de obligado cumplimiento. Es el grupo ideal y más simple de tratar. U ocurre, o se rechaza el producto.

?? **opcionales**: Una cierta prueba es opcional, siendo decisión del fabricante implementarla o no. Resultan casos de prueba parametrizados por valores que sólo se conocerán al ir a pasar las pruebas. Estos valores los proporcionará el fabricante. El operador introducirá los valores concretos para un cierto producto. Un cierto número de pruebas puede quedar anulado basándose en estos valores. Típicamente, las pruebas asociadas a cierto parámetro suelen estar agrupadas bajo el mismo propósito.

?? **indeterministas**: Es muy frecuente en software de comunicaciones, de sistemas distribuidos y otros que un cierto objetivo no se pueda alcanzar, sin que ello implique un error en el producto. Durante el proceso de elaboración de la batería de pruebas, combinando la especificación del sistema con los propósitos de prueba, es

perfectamente detectable este tipo de posibles comportamientos. Así, es muy sencillo incluir información adicional en el caso de prueba de forma que si en vez del comportamiento deseado se observa otro comportamiento igualmente posible, se emita un veredicto de *inconcluso*. Sólo comportamientos inadecuados llevarán a un veredicto de falla.

Sin embargo, debido a comportamientos no deterministas de la especificación, puede ocurrir que, al intentar ejecutar la prueba, el producto actúe de forma legal (según lo especificado) pero apartándose del propósito de la prueba. La prueba contempla estos indeterminismos dirigiendo la implementación a un estado estable pero, evidentemente, no se puede asegurar que el propósito ha sido cumplido. Por ello, se clasifican las pruebas dependiendo de si se puede asegurar el cumplimiento de su objetivo. Esta clasificación, ampliada de [Nicola] es:

- **MUST ACCEPT**: La especificación es determinista en el comportamiento referente al objetivo de la prueba, por lo que el producto debe aceptar, en todo momento, las ofertas de la prueba asociada.
- **MAY ACCEPT**: La especificación admite comportamientos que se apartan del objetivo de la prueba, por lo que la terminación de la prueba puede no asegurar el cumplimiento de su propósito.
- **MUST REJECT**: Igual que MUST, pero la intención es que se rechace.
- **MAY REJECT**: Igual que MAY, pero con la intención de que se rechace.

Ejecución de las Pruebas

La ejecución de las pruebas consiste en la aplicación de una batería de casos de prueba a un producto. Esta se realiza bajo control de un operador que introduce los parámetros necesarios, además de tener en cuenta los detalles de implementación del programa bajo pruebas.

Imposición versus Observación

La ejecución de una prueba conleva la aplicación de ciertos estímulos y la observación de reacciones asociadas. Por lo tanto, en una prueba habrá dos tipos de eventos: **imponibles** y **observables**. En software, los eventos imponibles serán las acciones que ejecutamos en el programa. Los eventos observables son los resultados esperados de cada instrucción (valores devueltos, efectos sobre variables, etc).

Seguimiento Sobre la Especificación de Referencia

El caso de prueba es el motor de ejecución de la prueba. Según se van observando eventos, se genera un informe detallado. Si llegamos a un resultado satisfactorio (PASA), poco más hay que decir de la prueba. Pero si llegamos a un resultado inconcluso o, aún más, si llegamos a una situación de bloqueo, hay que informar al fabricante de qué ha ocurrido. Obviamente, el producto es para el operador de pruebas una caja negra y no puede identificar errores dentro de él. Este es el problema del fabricante que tendrá que irse a su fábrica, identificar el error en su código, y corregirlo. Pero hay que indicarle exactamente qué requisito de la especificación se ha incumplido. Basta imprimir la traza y el estado final para proporcionar una valiosa información de qué aspecto de la especificación de referencia se incumplió, referido a ésta misma.

Bloqueo

En cualquier momento puede ocurrir cualquier evento no previsto o, más precisamente, que ninguno de los eventos previstos ocurran dentro del plazo marcado en el propósito de prueba. Cuando un temporizador salta decimos que se ha producido un *bloqueo*.

Un bloqueo puede ocurrir durante la ejecución del preámbulo o durante la ejecución del cuerpo de la prueba. En el primer caso se asigna un veredicto de *inconcluso*, siguiendo la costumbre de los centros de homologación y las recomendaciones de la ISO [ISO:9646].

Esta asignación es arbitraria y, usualmente, conlleva la repetición de la prueba un cierto número de veces. En cuanto pasa una vez, se considera pasada y no se repite más; pero si al cabo de un número predeterminado de intentos sigue fallando el preámbulo, se le asigna un veredicto final de inconcluso. Esto también es arbitrario; pero sigue siendo práctica habitual.

Muy diferente resulta la situación en la que el probador se bloquea durante la ejecución del cuerpo de la prueba. En estos casos, el veredicto es inequívocamente FALLO. Se emite un informe del estado de la especificación de referencia en el momento del bloqueo, y se rechaza el producto[1].

Pruebas correctas

Cada caso de prueba debe incluir información acerca de la consecución de su propósito. Por ello definimos como *prueba correcta* aquella que cumple las siguientes condiciones:

1. Debe contemplar un único propósito de prueba. De aquí se deriva que:
   - La prueba debe contemplar un solo caso de terminación en FALLA o PASA. El primer caso corresponderá a pruebas de rechazo, mientras que el segundo son pruebas de aceptación.
   - Si comportamientos aceptables en la especificación apartan la ejecución del propósito de la prueba podrán existir ramas de la prueba que terminen en INCONCLUSO.
2. La terminación de una prueba está limitada según su tipo. Así:
   - una prueba MUST ACCEPT debe consistir en una única traza terminante con etiqueta final PASA.
   - una prueba MAY ACCEPT debe contener más de una traza terminante, pero sólo una de ellas tendrá etiqueta PASA. Las demás tendrán forzosamente etiqueta INCONCLUSO.
   - una prueba MUST REJECT debe consistir en una única traza terminante con etiqueta final FALLA. Además se considera (por tener propósito único) que solo existirá un evento de rechazo. Todos la secuencia de eventos anteriores será considerada como preámbulo de la prueba.
   - una prueba MAY REJECT debe contener más de una traza terminante, pero sólo una de ellas con etiqueta FALLA. Igualmente que en el caso anterior, se considera que el cuerpo de la prueba comienza en el estado justamente anterior al evento a rechazar. El resto de las trazas deben terminar en INCONCLUSO.

[1] A efectos prácticos, puede que se intenten algunas pruebas más para aprovechar la sesión de homologación e intentar descubrir más errores antes de devolver el producto al fabricante; pero pase lo que pase con cualquier otra prueba, el producto será rechazado.

3. En cada estado de la prueba pueden existir un único tipo de eventos:

   - **imponibles**: un evento imponible es aquel que la prueba trata de forzar su ejecución por la implementación. Si ésta lo rechaza se considera que le ejecución ha sido bloqueada. Sólo puede existir un evento imponible en un estado dado.

   - **aceptables**: un evento aceptable es aquel que la prueba se limita a observar en su ejecución. Cuando el producto ejecuta uno de tales eventos la prueba se limita a evolucionar acordemente e intercambiando datos si fuera menester. Pueden existir varios eventos aceptables en un estado dado, debido a que se pueden obtener diferentes resultados.

Asignación de veredictos

La asignación de veredictos descrita en la norma ISO-9646 necesita de interpretación, puesto que va a depender, por un lado, de la parte de la prueba que se esté ejecutando y por otro, de que la prueba sea de rechazo o de aceptación.

Si la ejecución de la prueba se interrumpe en el preámbulo de la misma no se puede aseverar el cumplimiento de su propósito.

A continuación se presenta una tabla de asignación de veredictos que relaciona la terminación de la ejecución de la prueba (filas) con la clasificación de la misma (columnas).

	CLASIFICACIÓN			
	MUST ACCEPT	MAY ACCEPT	MUST REJECT	MAY REJECT
Bloqueo Preámbulo	Inconcluso	Inconcluso	Inconcluso	Inconcluso
Bloqueo Cuerpo	Falla	Inconcluso	Pasa	Pasa
Termina	Pasa	Falla	Falla	Inconcluso

En la primera columna se presentan todas las posibles terminaciones de la ejecución de una prueba. Las dos primeras filas se refieren al caso en que exista un bloqueo antes de terminar la prueba. Si se está ejecutando el preámbulo, el veredicto es INCONCLUSO. Si ocurre ya en el cuerpo principal de la prueba obtenemos normalmente un FALLO, excepto en el caso de prueba MUST REJECT, puesto que precisamente la intención de la prueba es que se rechace el evento que forma el cuerpo de la prueba.

Las tres últimas filas se refieren a la terminación de la prueba. Si se llega al final, dependiendo de cómo se terminó la ejecución y del objetivo de la prueba asignamos veredictos.

En el caso de MAY ACCET y MAY REJECT, aunque los bloqueos en el cuerpo son esperados, también se espera que alguna vez el MAY ACCEPT no se bloque en el cuerpo, y el MAY REJECT puede ser aceptado en el Cuerpo.

Prueba de Programas Orientados a Objetos

La programación Orientada a Objetos (POO), la más popular hoy en día, tiene ciertos elementos específicos que hacen que la prueba de estos programa ofrezca ciertas peculiaridades deberemos tener en cuenta. Muchos de estos elementos específicos, que suponen una gran

ventaja para el diseño y producción del software, ofrecen sin embargo mayores dificultades para una adecuada prueba del mismo.

La naturaleza de la POO

En un POO, la unidad básica es la Clase, en lugar del subprograma (función, módulo, etc.), lo que implica que todos los esfuerzos de prueba de unidades deban centrarse en la clase, sin embargo, y como luego veremos la clase, como elementos de pruebas no es uniforme. Podemos encontrar diversos tipos de clases, cada uno con diferentes propiedades desde el punto de vista de la prueba de las mismas.

La reutilización en el contexto de la POO es un elemento que se promueve, y que la propia naturaleza de los lenguajes OO soporta con mayor facilidad que en otras metodologías. Los componentes software se utilizarán y reutilizarán en varios contextos, algunos significativamente diferentes de los que el desarrollador tenía en mente, por lo que serán más necesarios los componentes robustos y bien probados. Sin embargo es bien conocido que el desarrollo de sistema reutilizables requiere mayor esfuerzo (de 2 a 4 veces), también en la parte de pruebas, que en los componentes que no se diseñan para su reutilización.

Algunos elementos específicos de la POO, van a tener gran influencia en la prueba de los programas, como luego veremos:

- **Encapsulado:** hay un problema fundamental de observabilidad.
- **Herencia:** nos enfrenta al problema de probar los programas modificados.
- **Polimorfismo:** introduce el problema de la indecidibilidad.

Problemas específicos de la POO

En este apartado vamos a revisar brevemente el efecto que las particularidades de la POO tienen sobre la forma de realizar las pruebas y los resultados que podemos esperar.

La clase como unidad de Pruebas

La utilización de la clase como unidad de pruebas ofrece una serie de diferencias respecto al uso tradicional del procedimiento o del módulo.

El primer elementos, que supone una gran ventaja, pero sobre todo desde le punto de vista de la calidad y en la depuración de los programas es la carencia de variables globales, que aparecen ahora encapsulados dentro de las correspondientes clases. No hay variables globales y los datos no son compartidos entre unidades.

Sin embargo otros aspectos no son tan positivos, como el hecho de que una Clase no es un componente que se pueda probar, debe ser probado mediante una instancia suya. Sin embargo, no todas las clases permiten la creación de objetos a partir de las mismas, como en el caso de las clases interface, o de las clases abstractas. Deberemos entonces utilizar algún tipo de mecanismo indirecto par buscar los posibles fallos en ese código.

Por otro lado, no se pueden probar las operaciones de una clase de forma aislada, por lo que deberemos abordar la prueba de combinaciones o secuencias de las mismas, intentando

diseñas lo mejor posibles las pruebas para que el objetivo de las mismas este lo más focalizado posible en la operación de nuestro interés. Sin embargo, todo objeto tiene un estado, por lo que no es posible reducir la prueba de un objeto a la prueba de cada operación de forma independiente

Encapsulado

Los mecanismos de encapsulado nos llevan al problema de la ocultación. Si bien desde el punto de vista del diseño y la codificación esto es un elemento de gran importancia para escribir código correcto, dificulta enormemente la prueba de los programas, dejando muchos elementos indecidibles.

Cuando se oculta información, el comportamiento de un objeto debe hacerse a través de sus operaciones. Para intentar tener acceso a estos datos, y poder con ello obtener unos mejores resultados con nuestras pruebas, se pueden implementar varias estrategias:

- Una primera estrategia es modificar la clase probada para añadir operaciones que proporcionen visibilidad de los elementos ocultos
- Una estrategia más refinada es definir estas operaciones en una clase descendiente, y probar esta clase derivada

Herencia

La herencia en la POO nos pone frente al problema clásico de probar lo ya probado (retesting en ingles). Cuando, cuanto y cómo debemos probar un software (una clase en este caso), que ya ha sido probada, pero que ha sufrido algunas (típicamente pocas) modificaciones.

Algunas de las propiedades heredades deben ser probadas de nuevo en el contesto de la clase derivada, pero:

- La herencia puede romper la encapsulación, por lo que las clases derivadas pueden, en algunos casos, alterar los elementos heredados.
- La redefinición implica en cualquier caso probar de nuevo los elementos redefinidos, ya que aún no existe una tecnología fiable aplicable a lenguajes como Java que permita reutilizar el trabajo hecho sobre la clase de la cual derivamos.

Polimorfismo

El polimorfismo, que mediante la “ligadura tardía” permite decidir el método específico que se va a aplicar, tiene como contrapartida, precisamente el hecho de que cuando planificamos las pruebas no conocemos exactamente el código que se va a ejecutar. Incluso no conocemos el código que se puede ejecutar en el futuro. Estos problemas están en su mayoría causados por la extensibilidad de jerarquía de clases:

- Como la prueba de una operación consiste en comprobar sus efectos cuando se ejecutan varias combinaciones de valores de los parámetros, un Conjunto de Pruebas debe contemplar todos los posibles casos de “ligadura”.
- No se puede planificar una prueba en la cual se contemplen todos los posibles valores de los parámetros, porque una jerarquías de clases puede extenderse libremente, salvo que se limite explícitamente.

La existencia de contenedores de objetos heterogéneos y el casting de clases hace que muchos métodos no se puedan probar con profundidad por las limitaciones anteriormente expuestas,

confiando en que futuras clases ofrecerán los mismos resultados que las que hemos usado como modelo durante las pruebas.

Aspectos Sicológicos y Organización del Trabajo

Parecen tonterías; pero pueden cambiar radicalmente el éxito de una fase de pruebas:

1. Probar es ejercitar un programa para encontrarle fallos. Jamás se debería probar un programa con el ánimo de mostrar que funciona; ése no es el objetivo.
2. Un caso de prueba tiene éxito cuando encuentra un fallo. Lo gracioso no es encontrar un caso en el que el programa funciona perfectamente. Eso es, simplemente, lo normal. Lo ideal es encontrar el caso en el que falla.
3. Las pruebas debe diseñarlas y pasarlas una persona distinta de la que ha escrito el código; es la única forma de no ser "comprensivo con los fallos". Hacer una "obra maestra" cuesta mucho esfuerzo y requiere gran habilidad. Encontrarle fallos a una "obra maestra" cuesta aún más esfuerzo y exige otro tipo de habilidad.
4. Las pruebas no pueden esperar a que esté todo el código escrito para empezar a pasarlas. Deben irse pasando pruebas según se va generando el código para descubrir los errores lo antes posible y evitar que se propaguen a otros módulos.
5. Si en un módulo (o sección de un programa, en general) se encuentran muchos fallos, hay que insistir sobre él. Es muy habitual que los fallos se concentren en pequeñas zonas.
6. Las pruebas pueden encontrar fallos; pero jamás demostrar que no los hay. Es como las brujas: nadie las ha visto; pero haberlas, haylas. Ningún programa (no trivial) se ha probado jamás al 100%.
7. Si se detecta un fallo aislado, puede bastar una corrección aislada. Pero si se detectan muchos fallos en un módulo, lo único práctico es desecharlo, diseñarlo de nuevo, y recodificarlo. La técnica de ir parcheando hasta que se pasan una serie de pruebas es absolutamente suicida y sólo digna del avestruz.

Conclusiones

Probar es buscarle los fallos a un programa.

La fase de pruebas absorbe una buena porción de los costes de desarrollo de software. Además, se muestra renuente a un tratamiento matemático o, simplemente, automatizado. Su ejecución se basa en metodología (reglas que se les dan a los encargados de probar) que se va desarrollando con la experiencia. Es tediosa, es un arte, es un trabajo que requiere una buena dosis de mala intención, y provoca difíciles reacciones humanas.

Aunque se han desarrollado miles de herramientas de soporte de esta fase, todas han limitado su éxito a entornos muy concretos, frecuentemente sólo sirviendo para el producto para el que se desarrollaron. Sólo herramientas muy generales como analizadores de complejidad, sistemas de ejecución simbólica, etc. han mostrado su utilidad en un marco más amplio. Pero al final sigue siendo imprescindible un artista humano que sepa manejarlas.

Bibliografía

?? [Myers79? Glenford J. Myers “El Arte de Probar el Software”, El Ateneo, 1983. “The Art of Software Testing” John Wiley & Sons, Inc. 1979

?? [Mynatt90? Barbee Teasley Mynatt “Software Engineering with Student Project Guidance”. Prentice-Hall International Editions, 1990

?? [Pressman87? Roger S. Pressman “Software Engineering: A Practitioner’s Approach” McGraw-Hill Intl. Eds. 1987

?? [Nicola84? Rocco de Nicola, M. Hennessy “Testing Equivalences for Processes” tcs. 1984 vol 34, pps: 83-133.

Glosario

Aunque he intentado utilizar traducciones razonables e intuitivas de los términos mas habitualmente utilizados, es bien cierto que lo más frecuente es que en la práctica nos encontremos la literatura en inglés. Esta segunda versión del glosario intenta cubrir la terminología anglosajona.

testing	pruebas
test oracle	oráculo
test harness	banco de pruebas
unit testing	pruebas de unidades
integration testing	pruebas de integración
acceptance testing	pruebas de aceptación
testing in the small	pruebas a pequeña escala
testing in the large	pruebas a escala industrial
incremental coding	codificación incremental
desk checking	pruebas de despacho
hand execution	ejecución manual
debugging	depuración
static testing	pruebas estáticas
dynamic testing	pruebas dinámicas
back-box testing	pruebas de caja negra
white-box testing	pruebas de caja blanca
data-structure-based testing	pruebas basadas en datos
coverage	cobertura
segment coverage	cobertura de segmentos
statement coverage	cobertura de sentencias
decision coverage	cobertura de decisiones
loop coverage	cobertura de bucles
logic-flow diagram	diagrama de flujo de lógica
equivalence partitioning	particiones de equivalencia
boundary testing	pruebas de casos límite
error-prone modules	módulos sospechosos
structure tests	pruebas estructurales
functional tests	pruebas funcionales
performance tests	pruebas de prestaciones
stress tests	pruebas de robustez
robustness tests	pruebas de robustez
transaction-flow diagram	diagrama de flujos de transacciones
alpha testing	pruebas a nivel alfa
beta testing	pruebas a nivel beta
verification	verificación
validation	validación
quality	calidad
regression testing	pruebas de regresión

Caso Práctico

Los ejemplos de pruebas de programas suelen irse a uno de dos extremos: o son triviales y no se aprende nada, o son tan enormes que resultan tediosos. El ejemplo elegido para esta

sección pretende ser comedido, a costa de no contemplar mas que un reducido espectro de casos.

Nos dan para probar un procedimiento

```
PROCEDURE Busca (C: CHAR; V: ARRAY OF CHAR): BOOLEAN;
```

A este procedimiento se le proporciona un caracter C y un array V de caracteres. El ARRAY debe estar ordenado alfabéticamente, en orden ascendente. El procedimiento devuelve TRUE si C está en V, y FALSE si no. Trabajamos en Modula-2.

Lo primero que hay que hacer es identificar clases de equivalencia sobre su interfaz:

C: CHAR.- El parámetro C está muy poco especificado. Sólo se dice que es un carácter, lo que queda de lo más ambiguo pues esto significa conjuntos diferentes dependiendo del ordenador.

V: ARRAY OF CHAR.- No se dice nada del conjunto de caracteres posibles (como en C), ni de las dimensiones límite del ARRAY. Tampoco se dice nada del criterio de ordenación.

resultado: BOOLEAN.- Éste si está perfectamente claro.

Para probar algo necesitamos saber más. La única forma es tener una charla con el que especificó la función y aclarar estos extremos. Todas estas aclaraciones deben quedar recogidas por escrito en una nueva versión de la especificación:

A este procedimiento se le proporciona un caracter C y un array V de caracteres. Se admitirá cualquier caracter de 8 bits de los representables en un PC con Modula-2. El ARRAY podrá tener entre 0 y 10.000 caracteres y estar ordenado alfabéticamente, en orden ascendente. El orden de los caracteres es el proporcionado por el Modula-2 sobre el tipo CHAR. Es admisible cualquier cadena de caracteres construida según el convenio de Modula-2 para este tipo de datos. El procedimiento devuelve TRUE si C está en V, y FALSE en caso contrario. Trabajamos en Modula-2.

Con estas explicaciones identificamos las siguientes clases de equivalencia

?? C: CHAR

  ?? Cualquier caracter

?? V: ARRAY OF CHAR

  ?? El ARRAY vacio.

  ?? Un ARRAY entre 1 y 10.000 elementos, ambos inclusive, ordenado.

  ?? Un ARRAY entre 1 y 10.000 elementos, ambos inclusive, desordenado.

?? resultado: BOOLEAN

  ?? TRUE

  ?? FALSE

Por último, cabe considerar combinaciones significativas de datos de entrada: que C sea el primero o el último del ARRAY.

1 Pruebas de caja negra: valores límite
  1.1 Buscar el caracter 'k' en el ARRAY ""
    Debe devolver FALSE.

1.2 Buscar el caracter 'k' en el ARRAY "k"
Debe devolver TRUE.
1.3 Buscar el caracter 'k' en el ARRAY "j"
Debe devolver FALSE.
1.4 Buscar el caracter 'k' en el ARRAY "kl"
Debe devolver TRUE.
1.5 Buscar el caracter 'k' en el ARRAY "jk"
Debe devolver TRUE.
1.6 Buscar el caracter 'k' en el ARRAY de 10.000 "a"
Debe devolver FALSE.

Vamos a olvidar de momento las posibles pruebas referentes a la ordenación del ARRAY.

2 Pruebas de caja negra: valores normales
2.1 Buscar el caracter 'k' en el ARRAY "abc"
Debe devolver FALSE.
2.2 Buscar el caracter 'k' en el ARRAY "jkl"
Debe devolver TRUE.

Para pasar a caja blanca necesitamos conocer el código interno:

```
 1       PROCEDURE Busca (C: CHAR; V: ARRAY OF CHAR):
BOOLEAN;
 2      VAR a, z, m: INTEGER;
 3    BEGIN
 4       a:= 0;
 5       z:= Length (V) - 1;
 6       WHILE (a <= z) DO
 7          m:= (a+z) DIV 2;
 8          IF V[m] = C THEN RETURN TRUE;
 9          ELSIF V[m] < C THEN a:= m+1;
10          ELSE z:= m-1;
11          END;
12       END;
13       RETURN FALSE;
14    END Busca;
```

Es laborioso; pero si nos molestamos en ejecutar todas las pruebas anteriores marcando por dónde vamos pasando sobre el código, nos encontraremos con que hemos ejecutado todas las sentencias con excepción de la rama de la línea 10. Para atacar este caso necesitamos un caso de prueba adicional de caja blanca

3 Pruebas de caja blanca:
3.1 Buscar el caracter 'k' en el ARRAY "l"
Debe devolver FALSE.

Con el conjunto de pruebas que llevamos hemos logrado una cobertura del 100% de segmentos y de condiciones. Respecto del bucle, la prueba 1.1 lo ejecuta 0 veces, y las demás pruebas 1 o más veces.

El conjunto de pruebas identificado se puede traducir en un banco de pruebas con el siguiente aspecto:

```
    IF      Busca ('k', "")     THEN WriteLine ("falla 1.1");
END;
    IF  NOT Busca ('k', "k")    THEN WriteLine ("falla 1.2");
END;
    IF      Busca ('k', "j")    THEN WriteLine ("falla 1.3");
END;
    IF  NOT Busca ('k', "kl")   THEN WriteLine ("falla 1.4");
END;
    IF  NOT Busca ('k', "jk")   THEN WriteLine ("falla 1.5");
END;
    IF      Busca ('k', aaaa)   THEN WriteLine ("falla 1.6");
END;
    IF      Busca ('k', "abc") THEN WriteLine ("falla 2.1");
END;
    IF  NOT Busca ('k', "jkl") THEN WriteLine ("falla 2.2");
END;
    IF      Busca ('k', "l")    THEN WriteLine ("falla 3.1");
END;
```

Aún podríamos pasar algunas pruebas más para comprobar la solidez del programa. Concretamente, sería bueno considerar qué ocurre si sobrepasamos el tamaño máximo de 10.000 caracteres o si el ARRAY estuviera desordenado. La especificación del módulo no dice nada de esto, por lo que el análisis del resultado es vidrioso. Sobre el código concreto podemos apreciar que el tamaño del ARRAY puede llegar hasta el máximo entero soportable por la implementación de Modula-2 que estemos usando. Sobrepasado este límite se puede producir un error de asignación fuera de rango en la línea 5. Por otra parte, si el ARRAY está desordenado, el resultado es arbitrario, aunque la función siempre termina devolviendo TRUE o FALSE.