Prueba de Programas

El presente texto está basado en el material de José A. Mañas, denominado *Prueba de Programas*, http://www.lab.dit.upm.es/~lprg/material/apuntes/pruebas/testing.htm

1. Introducción

Como hemos visto, uno de los últimos pasos para solucionar un problema es la ejecución y prueba. Nos centraremos aquí en la prueba de programas.

Aunque parezca trivial, la cantidad de tiempo y esfuerzo que demanda la prueba de un programa es considerable. Se estima que la mitad del esfuerzo de desarrollo de un programa, tanto en tiempo como en costo, se invierte en esta etapa.

Especialistas en Ingeniería de Software estiman que por cada hora que se le dedica a la prueba (testing) de un programa (sistema) se ahorran diez horas de soporte o mantenimiento, además de ganar en calidad del producto. Cada error detectado y corregido supone un importante descenso en el tiempo de corrección de un programa tiempo después de haberlo entregado al usuario final.

1.1. ¿Qué es probar?

Es muy común que los programas tengan errores. Más aún si no somos disciplinados a la hora de construir un programa. La etapa de prueba es la encargada de encontrar dichos errores. El objetivo específico de la prueba es encontrar la mayor cantidad de errores posibles. Sin embargo, la prueba puede mostrar la presencia de errores, pero nunca su ausencia (E. W. Dijkstra). Es decir que el hecho de no encontrar más errores no garantiza que no los halla. Por ello se dice que probar un programa es usarlo con la peor intención a fin de encontrarle fallas (G. J. Myers).

1.2. Prueba Exhaustiva

La prueba ideal de un programa sería exponerlo bajo todas las situaciones posibles. Pero lamentablemente esto es imposible.

Dado que tanto los algoritmos como los programas son un conjunto finito de instrucciones podríamos pensar que el número de pruebas posibles también es finito. Sin embargo esto deja de ser cierto en cuanto entran en juego los ciclos. Y es precisamente en los ciclos donde se producen la mayor cantidad de errores. 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 someterlo a todas las posible variaciones de los datos de entrada, tanto si son válidos como si no lo son.

Sobre esta premisa de imposibilidad de alcanzar la perfección, hay que buscar alternativas que sean aceptables para encontrar errores.

1.3. Organización

Hay múltiples puntos de vista respecto a la prueba de programas. Por ello, sin desmerecer otras visiones, para el desarrollo del tema utilizaremos uno de los ejes clásicos en Ingeniería de Software.

La prueba de programas puede dividirse en diferentes etapas denominadas: **prueba de unidades**, **prueba de integración** y **prueba de aceptación**. Desarrollaremos con mayor profundidad la **prueba de unidades** puesto que es la que más nos interesa en nuestra materia ya que estamos desarrollando programas de pequeña y pequeña-mediana escala.

- UNIDADES
  - CAJA BLANCA
    - Cobertura:
      - de sentencias
      - de ramas
      - de condición/decisión
      - de ciclos
  - 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 un programa.

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 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-down), como el que utilizamos nosotros, se empieza a probar por los módulos más generales; mientras que en un diseño ascendente (botton-up) se empieza a probar por los módulos de base.

El planteamiento descendente tiene la ventaja de estar siempre pensando en términos de la funcionalidad global; pero también tiene el inconveniente de que para cada prueba hay que "inventarse" algo sencillito (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".

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 el programa en producción.

2. Prueba de Unidades

¿Cómo se prueban módulos sueltos?

Normalmente cabe distinguir una fase informal antes de entrar en la fase de pruebas propiamente dicha. La fase informal la lleva a cabo el propio codificador en su escritorio, y consiste en ir ejecutando el código para convencerse de que funciona. De allí que ese tipo de pruebas reciban el nombre de **pruebas de escritorio**. Esta fase suele consistir en pequeños ejemplos que se intentan ejecutar.

Más adelante, cuando estamos convencidos de que el módulo soluciona el problema para el cual lo hemos creado, se ingresa 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.

2.1. Caja blanca

También llamadas pruebas estructurales, en estas pruebas estamos siempre observando el código 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 del código que hemos cubierto con la prueba.

Hay diferentes posibilidades de definir la cobertura, las cuales analizaremos a continuación.

Cobertura de sentencias

Como el número de sentencias de un programa es finito, se puede diseñar un plan de pruebas que vaya ejecutando 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.

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 mucha frecuencia que los programas contienen código muerto o inalcanzable. Puede ser que un 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 sentencias es engañosa en presencia de sentencias opcionales. Por ejemplo:

IF Condicion
THEN EjecutaEsto
SINO EjecutaAquello;

Desde el punto de vista de cobertura de sentencias, 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 porque si no lo fuera el IF sobraría.

Para afrontar estos casos, se plantea un refinamiento de la cobertura de sentencias 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.

Nótese que si lográramos una cobertura de ramas del 100%, esto llevaría implícita una cobertura del 100% de los sentencias, pues toda sentencia está en alguna rama.

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 EjecutaEsto
SINO EjecutaAquello;

Las condiciones 1 y 2 pueden tomar 2 valores cada una, dando lugar a 4 combinaciones posibles. 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 Condición1.

Para afrontar esta problemática se define un criterio de cobertura de condición/decisión que divide 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 aun 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 equivocamos 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.

Cobertura de ciclos

Este tipo de cobertura alcanzaremos a comprenderla luego de analizar el tema de **composición iterativa**.

Los ciclos no son más que segmentos controlados por decisiones. Así, la cobertura de ramas cubre plenamente la esencia de los ciclos. Pero eso es simplemente la teoría, pues la práctica descubre que los ciclos son una fuente inagotable de errores, todos triviales, algunos mortales. Un ciclo se ejecuta un 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 ciclo que se ejecuta 1 vez de más o de menos. Analizaremos este tema directamente sobre los ciclos que propone Pascal.

Para un ciclo de tipo WHILE hay que pasar 3 pruebas

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

Para un ciclo de tipo REPEAT hay que pasar 2 pruebas

1. 1 ejecución
2. más de 1 ejecución

Los ciclos 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 con ejecutarlos 1 vez.

No obstante, conviene no engañarse con los ciclos FOR y examinar su contenido. Si dentro del ciclo 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 ciclo FOR con trampa.

Y en la práctica ¿qué hago?

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

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.

2.2. Caja negra

También llamadas pruebas funcionales, las pruebas de caja negra se centran en lo que se espera de un módulo, es decir, intentan encontrar casos en que el módulo no respecta su especificación. Por ello se denominan pruebas funcionales, y el testeador 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 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 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 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:

1. números menores que 1
2. números entre 1 y 31
3. 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 elegir 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 "frontera" (o valores límite) 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.

3. Pruebas de Integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema, es decir un programa de mediana o gran dimensión. 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).

4. 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.

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.

Muchas veces se aplican las 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 fallas (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. La experiencia muestra que estas prácticas son muy eficaces.

5. 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 analizan el diseño del sistema, leen el programa línea a línea y piden explicaciones de todo lo que no está claro. Puede que simplemente falte un comentario explicativo, que se haya tomado una decisión de diseño incorrecta, que detecten un error auténtico, o que simplemente el código sea muy complejo de entender y/o explicar. Para un sistema complejo pueden hacer falta muchas sesiones.

Esta técnica 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.

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).

En función de ello, probablemente sea razonable comenzar la fase de pruebas con una serie de casos elegidos al azar. Esto pondrá de manifiesto los errores más gruesos. No obstante, pueden permanecer ocultos errores más finos que sólo se detectan mediante datos de entrada muy precisos.

Si el programa es poco crítico puede que esto sea suficiente, pero si se trata de una aplicación con riesgo para vidas humanas es insuficiente desde todo punto de vista.

Solidez (robustness testing)

Se prueba la capacidad del sistema para salir de situaciones 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 se deben probar los siguientes extremos:

- órdenes correctas, todas y cada una
- órdenes con defectos de sintaxis
- órdenes correctas, pero en orden incorrecto, o fuera de lugar
- la orden nula (línea vacia, 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]

Stress (stress testing)
Se usa para saber hasta que punto soporta un sistema, bien por razones internas (cuantos datos podrá procesar), bien externas (es capaz de trabajar con un disco al 90% o una carga de la CPU del 90, etc.)

Prestaciones (performance testing)
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, etc. 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 computadoras 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". Un ejemplo de ello fueron las pruebas de homologación para el año 2000.

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 éstas 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.

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.

6. Depuración (debugging)

Casi todos los compiladores suelen llevar asociada la posibilidad de ejecutar un programa paso a paso, permitiéndole al operador conocer dónde está en cada momento, y cuánto valen las variables. El compilador de Turbo Pascal que nosotros utilizamos permite realizar la depuración de programas mediante varias herramientas tales como Step Over (F8), Trace into (F7), Debug\Watchs, etc.

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.

7. 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?

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.

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

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.
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.

8. Conclusiones

Probar es buscarle los fallas a un programa.

La fase de pruebas absorbe una buena porción de los costos de desarrollo de software. Además, aún es bastante complejo un tratamiento matemático-lógico 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.

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 y medidores de cobertura han mostrado su utilidad en un marco más amplio. Pero al final sigue siendo imprescindible una persona que sepa manejarlas.

9. Bibliografia

- José A. Mañas. Prueba de Programas. 1994. http://www.lab.dit.upm.es/~lprg/material/apuntes/pruebas/testing.htm
- Glenford J. Myers. El Arte de Probar el Software (The Art of Software Testing) El Ateneo, 1983 (John Wiley & Sons, Inc. 1979).
- Barbee Teasley Mynatt. Software Engineering with Student Project Guidance Prentice-Hall International Editions, 1990.
- Boris Beizer. Software Testing Techniques Van Nostrand Reinhold (N.Y.) 2a ed. 1990.
- Roger S. Pressman. Software Engineering: A Practitioner's ApproachMcGraw-Hill Intl. Eds. 1987.