Tema 6.- Pruebas del Software

6.1.- Introducción.

Una de las características típicas del desarrollo de software basado en el ciclo de vida es la realización de controles periódicos, normalmente coincidiendo con los hitos de los proyectos o la terminación de documentos. Estos controles pretenden una evaluación de la calidad de los productos generados (especificación de requisitos, documentos de diseño, etc.) para poder detectar posibles defectos cuanto antes. Sin embargo, todo sistema o aplicación, independientemente de estas revisiones, debe ser probado mediante su ejecución controlada antes de ser entregado al cliente. Estas ejecuciones o ensayos de funcionamiento, posteriores a la terminación del código del software, se denominan habitualmente pruebas.

Las pruebas constituyen un método más (junto a las revisiones de los productos que preceden al código en el ciclo de vida) para poder verificar y validar el software. Se puede definir la verificación como: "El proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al principio de dicha fase". Por ejemplo, verificar el código de un módulo significa comprobar si cumple lo marcado en la especificación de diseño donde se describe. Por otra parte, la validación es: "El proceso de evaluación del sistema o de uno de sus componentes durante o al final del desarrollo para determinar si satisface los requisitos especificados". Así, validar una aplicación implica comprobar satisface los requisitos marcados por el usuario. Podemos recurrir a la clásica explicación informal de Boehm de estos conceptos:

1. **Verificación**: ¿estamos construyendo correctamente el producto?

2. **Validación**: ¿estamos construyendo el producto correcto?

Como hemos dicho, las pruebas permiten verificar y validar el software cuando ya está en forma de código ejecutable. A continuación, expondremos algunas definiciones de conceptos relacionados con las pruebas.

6.1.1.- Definiciones

Las siguientes definiciones son algunas de las recogidas en el diccionario IEEE en relación a las pruebas [IEEE. 1990], que complementamos con otras más informales:

1. **Pruebas** *(test*): «una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto». Para Myers, probar (o la prueba) es el «proceso de ejecutar un programa con el fin de encontrar errores». EI nombre «prueba», además de la actividad de probar, se puede utilizar para designar «un Conjunto de casos y procedimientos de prueba».

2. **Caso de prueba** *(test case):* «un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito». También se puede referir a la documentación en la que se describen las entradas, condiciones y salidas de un caso de prueba.

3. **Defecto** *(defect, fault, «bug»)*: «una incorrección en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa».

4. **Fallo** *(failure):* «La incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados».

5. **Error** *(error):* tiene varias acepciones:

    1. La diferencia entre un valor calculado, observado o medido y el valor verdadero, especificado o teóricamente correcto. Por ejemplo, una diferencia de dos centímetros entre el valor calculado y el real.
    2. Un defecto. Por ejemplo, una instrucción incorrecta en un programa.
    3. Un resultado incorrecto. *Por* ejemplo, un programa ofrece como resultado de la raíz cuadrada de 36 el valor 7 en vez de 6.
    4. Una acción humana que conduce a un resultado incorrecto (una metedura de pata: *mistake).* Por ejemplo, que el operador o el programador pulse una tecla equivocada.

Nosotros desecharemos las acepciones 2 y 3, ya que coinciden con las definiciones de defecto y fallo, para evitar equívocos.

6.2.- Filosofía de las Pruebas del Software

Las especiales características del software (ausencia de existencia física, ausencia de leyes que rijan su comportamiento, gran complejidad, etc.) hacen aún más difícil la tarea de probarlo en relación con otros productos industriales. Como veremos posteriormente, la prueba exhaustiva del software es impracticable: no se pueden probar todas las posibilidades de su funcionamiento incluso en programas pequeños y sencillos. Además, existen prejuicios que pueden perjudicar a las pruebas. Todos estos factores obligan a estudiar cuál es la mejor actitud o filosofía a seguir en esta actividad.

Hay que recordar (MYERS, 1979], ante todo, que el objetivo de las pruebas es la detección de defectos en el software y que descubrir un defecto debería considerarse el éxito de una prueba. Se trata de una actividad a posteriori, de detección, y no de prevención, de problemas en el software. El problema es que, tradicionalmente, existe el mito de la ausencia de errores en el buen profesional: si fuéramos realmente capaces y empleáramos las técnicas más sofisticadas, no existirían los defectos en el software. Por lo tanto, un defecto implica que somos malos profesionales y que debemos sentirnos culpables. Si una prueba revela uno de estos problemas, implica la constatación de un fracaso del desarrollador

Sin embargo, la realidad es muy distinta. Todo el mundo comete errores (errare humanum est) y es de sabios rectificar. Las pruebas permiten la rectificación en el software. La mayoría de los estudios revelan que los mejores programadores incluyen una cierta media de defectos por cada 1.000 líneas de código. Los defectos no son siempre el resultado de la negligencia, sino que en su aparición influyen múltiples factores (por ejemplo, la mala comunicación entre los miembros del equipo que da lugar a malentendidos en los requisitos pedidos; así, quizás sólo con telepatía no se cometen

errores). Por todo ello, el descubrimiento de un defecto significa un éxito para a mejora de la calidad al igual que, a pesar de lo incómodo que resulte, la detección de un problema de salud en un análisis médico se considera un éxito para lograr la curación del paciente.

Davis y Myers proponen una serie de recomendaciones para las pruebas. Davis propone[1]:

1. A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente. Como hemos visto, el objetivo de las pruebas de software es descubrir errores. Se entiende que los defectos más graves (desde el punto de vista del cliente) son aquellos que impiden al programa cumplir sus requisitos.

2. Las pruebas deberían planificarse mucho antes de que empiecen. La planificación de las pruebas puede comenzar tan pronto como esté completo el modelo de requisitos. La definición detallada de los casos de prueba puede empezar tan pronto como el modelo de diseño se ha consolidado. Por tanto, se pueden planificar y diseñar algunas pruebas antes de generar ningún código.

3. El 80% de los errores surgen al hacer un seguimiento del 20% de los módulos del software (Principio de Pareto). El problema, por supuesto consiste en aislar los módulos sospechosos y probarlos concienzudamente.

4. Las pruebas tendrían que hacerse de lo pequeño hacia lo grande. Las primeras pruebas planeadas y ejecutadas se centran generalmente en módulos individuales del programa. A medida que avanzan las pruebas, desplazan su punto de mira en un intento de encontrar errores en grupos integrados de módulos y finalmente en el sistema entero.

5. No son posibles pruebas exhaustivas. El número de permutaciones de caminos para incluso un programa de tamaño moderado es excepcionalmente grande. Por este motivo, es imposible ejecutar todas las combinaciones de caminos durante las pruebas. Es posible, sin embargo, cubrir adecuadamente la lógica del programa y asegurarse de que se han aplicado todas las condiciones en el diseño a nivel de componente.

6. Las pruebas, para ser más efectivas, deberían de ser realizadas por un equipo independiente del equipo de desarrollo del software. Por “más eficaces” queremos referirnos a pruebas con más alta probabilidad de encontrar errores (el objetivo principal de las pruebas). El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas. Esta actitud inadecuada lleva a realizar pruebas menos rigurosas de lo que sería deseable. Además, es normal que las situaciones que ha olvidado considerar al crear el programa (por ejemplo, no pensar en cómo tratar un fichero de entrada vacío) queden de nuevo olvidadas al escribir casos de prueba. Lo ideal sería que probara el software el peor

[1] Pressman 5ª Edición, pp282

enemigo de quien lo ha construido, ya que así se aseguraría una búsqueda implacable de cualquier error cometido.

Por su parte las recomendaciones de G. J. Myers para las pruebas son las siguientes:

1. Cada caso de prueba debe definir el resultado de salida esperado. Este resultado esperado es el que se compara con el realmente obtenido de la ejecución en la prueba. Las discrepancias entre ambos (errores) se consideran síntomas de un posible defecto en el software.

2. Se debe inspeccionar a conciencia el resultado de cada prueba para así, poder descubrir posibles síntomas de defectos. Lamentablemente, es frecuente pasar por alto síntomas bastante claros. Esto invalida todo el esfuerzo realizado en la planificación, diseño y ejecución de pruebas.

3. Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados. Es frecuente observar una tendencia a centrarse en lo esperado y lo válido.

4. Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):
   - Probar si el software no hace lo que debe hacer.
   - Probar si el software hace lo que no debe hacer, es decir, si provoca efectos secundarios adversos,

5. Se deben evitar los casos desechables, es decir, los no documentados ni diseñados con cuidado (por ejemplo, los que se teclean sobre la marcha), ya que suele ser necesario probar una y otra vez el software hasta que queda libre de defectos. No documentar o guardar los casos significa repetir constantemente el diseño de casos de prueba.

6. No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas. Hay que asumir que siempre hay defectos (ya están cuantificadas las tasas habituales de defectos de los mejores desarrolladores profesionales de software y no son cero) y que hay que detectarlos. Las estadísticas confirman que, prácticamente, el 40% del esfuerzo de desarrollo se consume en pruebas y depuración.

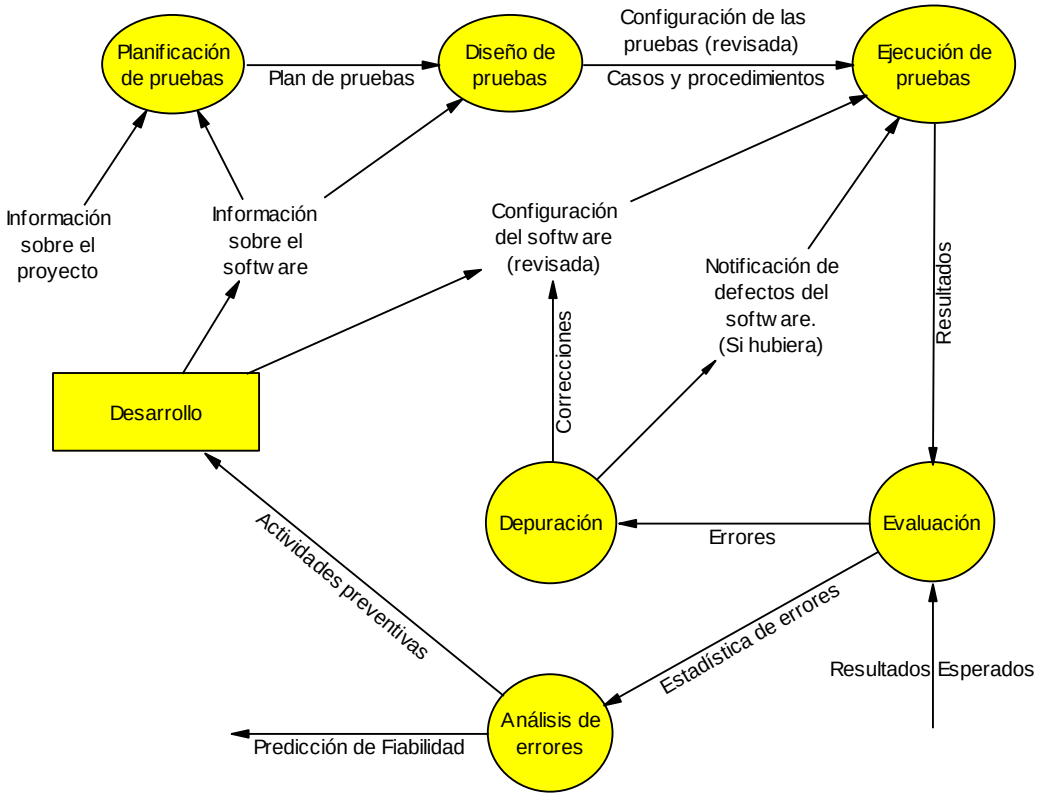
7. Las pruebas son una tarea tanto o más creativa que el desarrollo de software. Siempre se han considerado las pruebas como una tarea destructiva y rutinaria. No obstante, no existen técnicas rutinarias «como veremos para el diseño de pruebas y hay que recurrir al ingenio para alcanzar un buen nivel de detección de defectos con los recursos disponibles. Myers habla en su famoso libro [MYERS. 1979] del «arte de las pruebas». También se suele decir que «si cree que la construcción del programa ha sido difícil, aún no ha visto nada».

A los principios de Myers también es preciso añadir el principio de pareto y de que el grupo de pruebas tiene que ser independiente del de programación que ya recogíamos con Davis y que Myers repite.

Por lo tanto, la filosofía más adecuada para las pruebas consiste en planificarlas y diseñarlas de forma sistemática para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo. Así, debemos recordar que «un buen caso de prueba es aquel que tiene una gran probabilidad de encontrar un defecto no descubierto aún» y que «el éxito de una prueba consiste en detectar un defecto no encontrado antes». El contraste con la visión tradicional de las pruebas («vamos a probar un par de opciones para comprobar que funciona y ya está») es radical.

6.3.- El Proceso de Prueba

En la Figura se puede ver una representación del proceso completo relacionado con las pruebas basada en parte en el estándar y en parte en Pressman.



El proceso de prueba comienza con la generación de un plan de pruebas en base a la documentación sobre el proyecto y la documentación sobre el software a probar. A partir de dicho plan se entra en detalle diseñando pruebas específicas basándose en la documentación del software a probar. Una vez detalladas las pruebas (especificaciones de casos y de procedimientos) se toma la configuración del software (revisada, para confirmar que se trata de la versión apropiada del programa) que se va a probar para ejecutar sobre ella los casos. En algunas situaciones, se puede tratar de reejecuciones de pruebas, por lo que es conveniente tener constancia de los defectos ya detectados aunque aún no corregidos. A partir de los resultados de salida, se pasa a su evaluación mediante comparación con la salida esperada. A partir de ésta, se pueden realizar dos actividades:

1. La depuración (localización y corrección de defectos).
2. El análisis de la estadística de errores.

La depuración puede corregir o no los defectos. Si no consigue localizarlos, puede ser necesario realizar pruebas adicionales para obtener más información. Si se corrige un defecto, se debe volver a probar el software para comprobar que el problema está resuelto.

Por su parte, el análisis de errores puede servir para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y mejorar los procesos de desarrollo.

6.4.- Técnicas de diseño de casos de prueba

El diseño de casos de prueba está totalmente mediatizado por la imposibilidad de probar exhaustivamente el software. Pensemos en un programa muy sencillo que sólo suma dos números enteros de dos cifras (del O al 99). Si quisiéramos probar, simplemente, todos los valores válidos que se pueden sumar, deberíamos probar 10.000 combinaciones distintas de cien posibles números del primer sumando y cien del segundo. Pensemos que aún quedarían por probar todas las posibilidades de error al introducir los datos (por ejemplo, teclear una letra en vez de un número). La conclusión es que, si para un programa tan elemental debemos realizar tantas pruebas, pensemos en lo que supondría un programa medio.

En consecuencia, las técnicas de diseño de casos de prueba tienen como objetivo conseguir una confianza aceptable en que se detectarán los defectos existentes (ya que la seguridad total sólo puede obtenerse de la prueba exhaustiva, que no es practicable) sin necesidad de consumir una cantidad excesiva de recursos (por ejemplo, tiempo para probar o tiempo de ejecución). Toda la disciplina de pruebas debe moverse, por lo tanto, en un equilibrio entre a disponibilidad de recursos y la confianza que aportan los casos para descubrir los defectos existentes. Este equilibrio no es sencillo, lo que convierte a las pruebas en una disciplina difícil que está lejos de parecerse a la imagen de actividad rutinaria que suele sugerir.

Ya que no se pueden probar todas las posibilidades de funcionamiento del software, la idea fundamental para el diseño de casos de prueba consiste en elegir algunas de ellas que, por sus características, se consideren representativas del resto. Así, se asume que, si no se detectan defectos en el software al ejecutar dichos casos, podemos tener un cierto nivel de confianza (que depende de la elección de los casos) en que el programa no tiene defectos. La dificultad de esta idea es saber elegir los casos que se deben ejecutar. De hecho, una elección puramente aleatoria no proporciona demasiada confianza en que se puedan detectar todos los defectos existentes. Por ejemplo, en el caso del programa de suma de dos números, elegir cuatro pares de sumandos al azar no aporta mucha seguridad a la prueba (una probabilidad de 4/10000 de cobertura de posibilidades). Por eso es necesario recurrir a ciertos criterios de elección que veremos a continuación.

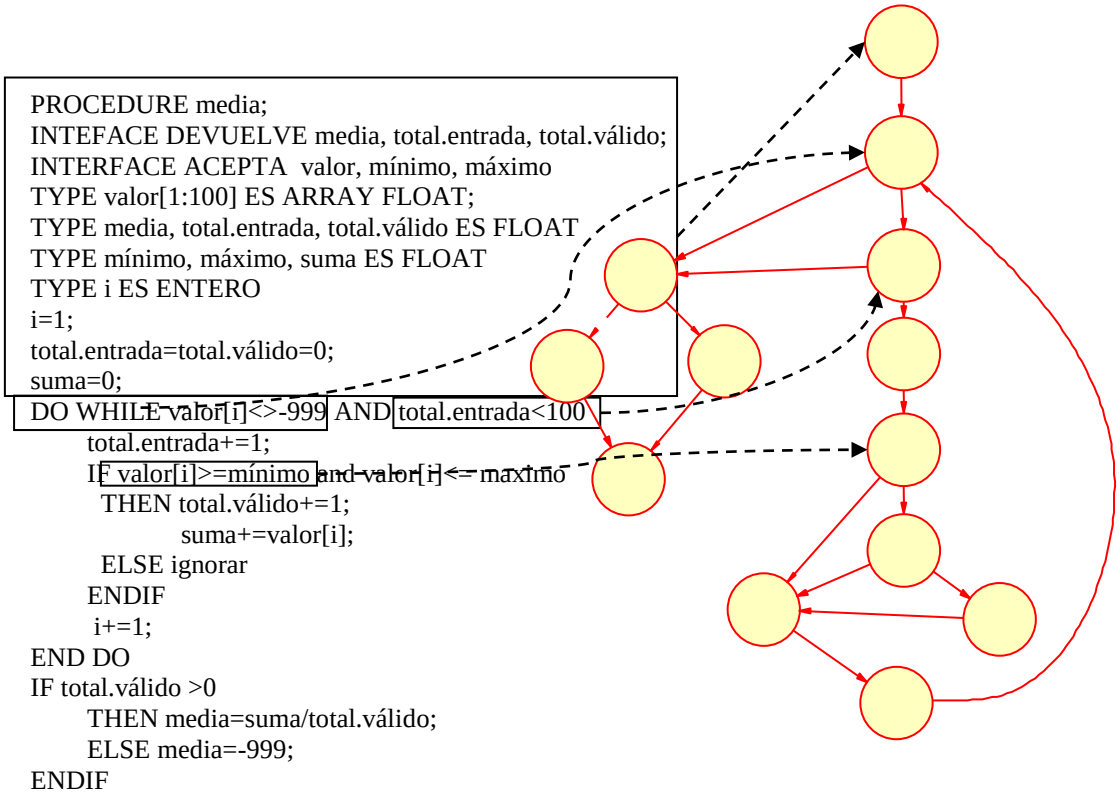
Existen dos enfoques principales para el diseño de casos:

1. El enfoque estructural o de caja blanca. Consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución, a través de las instrucciones del código, que puedan trazarse.

2. El enfoque funcional o de caja negra. Consiste en estudiar la especificación de las funciones, la entrada y la salida, para derivar los casos. Aquí, la prueba ideal del software consistiría en probar todas las posibles entradas y salidas del programa. La prueba de entradas y salidas supone la búsqueda de casos de prueba que admite dos posibilidades
   a. Búsqueda estructurada: que propone la búsqueda sistemática de un reducido número de pruebas que sean representativas del mayor número posible de situaciones de funcionamiento del sistema
   b. Búsqueda aleatoria que consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba. La prueba exhaustiva consistiría en probar todas las posibles entradas al programa.

Estos enfoques no son excluyentes entre sí, ya que se pueden combinar para conseguir una detección de defectos más eficaz. Veremos a continuación Una presentación de cada uno de ellos.

6.5.- Pruebas Estructurales



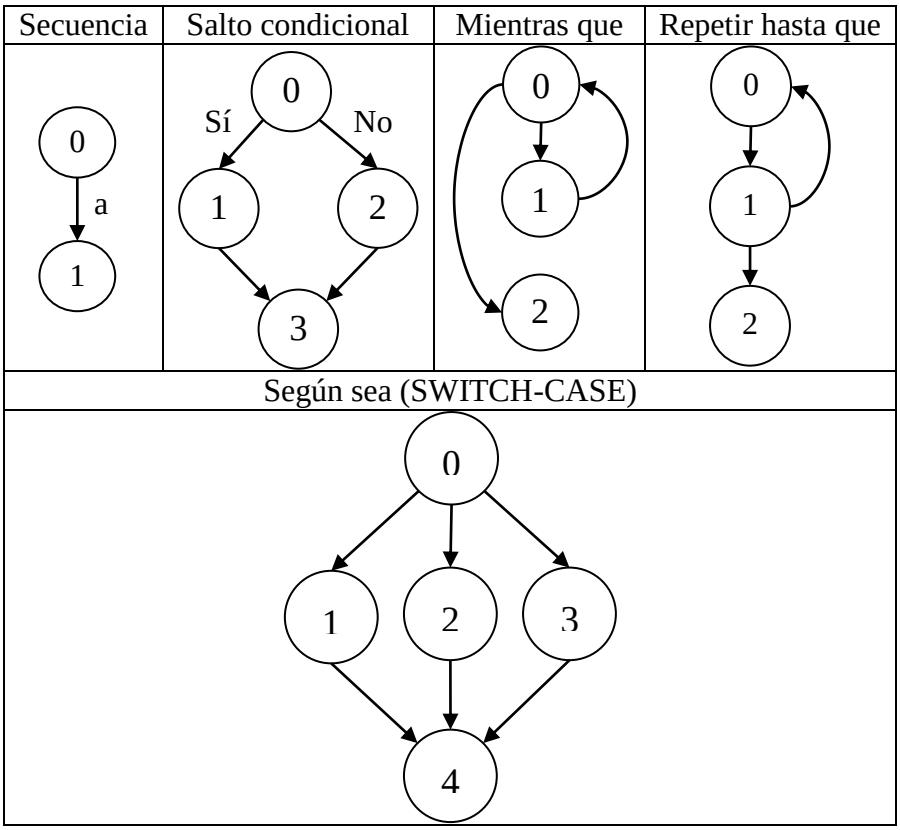
Como hemos dicho, las pruebas exhaustivas son impracticables. Podemos recurrir al clásico ejemplo de Myers de un programa de 50 líneas con 25 sentencias IF en serie, en el que el número total de caminos contiene *33,5* millones de secuencias potenciales (contando dos posibles salidas para cada IF tenemos 2^{25} posibles caminos). El diseño de casos tiene que basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable de descubrir un defecto, y para ello se utilizan los llamados criterios de cobertura lógica. Antes de pasar a examinarlos, conviene señalar que estas técnicas no requieren el uso de ninguna representación gráfica específica del software, aunque es

habitual tomar como base los llamados diagramas de flujo de control *(flowgraph charts flowcharts).* Como ejemplo de diagrama de flujo junto al código correspondiente podemos ver la figura siguiente

Para dibujar el grafo de flujo de un programa es recomendable seguir los siguientes pasos:

1. Señalar sobre el código cada condición de cada decisión (por ejemplo, en la figura valor[i]<>-999 y total.entrada<100 son condiciones distintas del primer bucle) tanto en sentencias IF-THEN y SWITCH-CASE como en los bucles FOR, DO o WHILE.

2. Agrupar el resto de las sentencias en secuencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas que mostramos a continuación.



3. Numerar tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es recomendable alterar el orden en el que aparecen las condiciones en una decisión multicondicional, situándolas en orden decreciente de restricción (primero, las más restrictivas). El objetivo de esta alteración es facilitar la derivación de casos de prueba una vez obtenido el grafo. Es conveniente identificar los nodos que representan condiciones asignándoles una letra y señalar cuál es el resultado que provoca la ejecución de cada una de las aristas que surgen de ellos (por ejemplo, si la condición x es verdadera o falsa).

Una posible clasificación de criterios de cobertura lógica es la que se ofrece abajo (MYERS, 1979). Hay que destacar que los criterios de cobertura que se ofrecen están en orden de exigencia y, por lo tanto, de coste económico. Es decir, e1 criterio de cobertura de sentencias es el que ofrece una menor seguridad de detección de defectos, pero es el que cuesta menos en número de ejecuciones del programa.

1. **Cobertura de sentencias**. Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.

2. **Cobertura de decisiones**. Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso. En general, una ejecución de pruebas que cumple la cobertura de decisiones cumple también la cobertura de sentencias.

3. **Cobertura de condiciones**. Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte ti valor verdadero al menos una vez y el falso al menos una vez. No podemos asegurar que si se cumple la cobertura de condiciones se cumple necesariamente la de decisiones.

4. **Criterio de decisión/condición**. Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.

5. **Criterio de condición múltiple**. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea (por ejemplo, según se ejecuta en el procesador se podría considerar que cada decisión multicondicional se descompone en varias decisiones unicondicionales). Es decir, una decisión como IF (a=1) AND (c=4) THEN se convierte en una concatenación de dos decisiones: IF (a=l) y IF (c=4). En este caso. debemos conseguir que todas las combinaciones posibles de resultados (verdadero/falso) de cada condición en cada decisión se ejecuten al menos una vez.

La cobertura de caminos (secuencias de sentencias) es el criterio más elevado: cada uno de los posibles caminos del programa se debe ejecutar al menos una vez. Se define camino como la secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final. Como hemos visto, el número de caminos en un programa pequeño puede ser impracticable para las pruebas. Para reducir el número de caminos a probar, se habla del concepto de camino de prueba *(test path):* un camino del programa que atraviesa, como máximo, una vez el interior de cada bucle que encuentra. La idea en la que se basa consiste en que ejecutar un bucle más de una vez no supone una mayor seguridad de detectar defectos en él. Sin embargo, otros especialistas [HUANG, 1979] recomiendan que se pruebe cada bucle tres veces: una sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces. Esto último es interesante para comprobar cómo se comporta a partir de los valores de datos procedentes, no del exterior del bucle (como en la primera iteración), sino de las operaciones de su interior. Saber cuál es el número de caminos del grafo de un programa ayuda a planificar las pruebas y a asignar recursos a las mismas, ya que indica el número de ejecuciones necesarias. También sirve de comprobación a la hora de enumerar los caminos.

Los bucles constituyen el elemento de los programas que genera un mayor número de problemas para la cobertura de caminos. Su tratamiento no es sencillo ni siquiera adoptando el concepto de camino de prueba. Pensemos, por ejemplo, en el caso de varios bucles anidados o bucles que fijan valores mínimo y máximo de repeticiones.

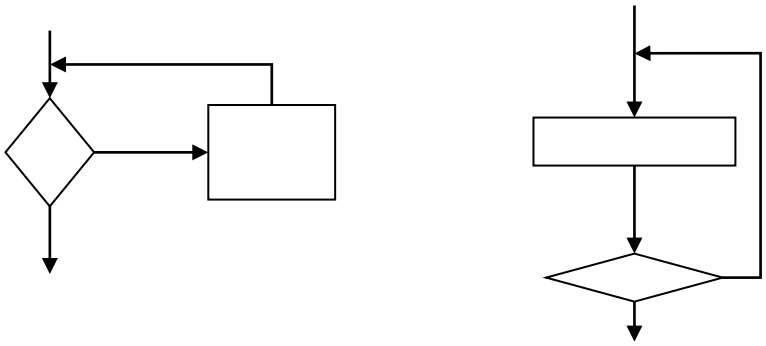
Prueba de bucles

Los bucles son la piedra angular de la inmensa mayoría de los algoritmos implementados en software. Y sin embargo. les prestamos normalmente poca atención cuando llevamos a cabo las pruebas del software.

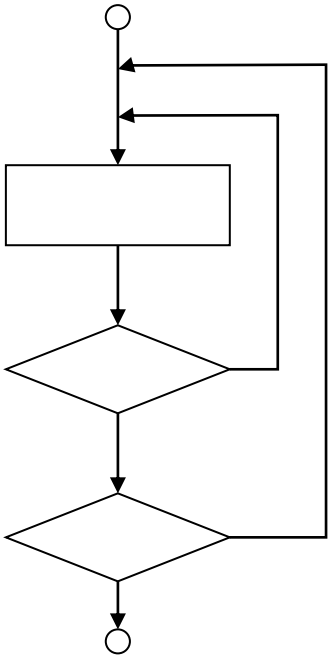
La prueba de bucles es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles. Se pueden definir cuatro clases diferentes de bucles: bucles simples, bucles concatenados, bucles anidados y bucles no estructurados.

Bucles simples. A los bucles simples se les debe aplicar el siguiente conjunto de pruebas, donde n es el número máximo de pasos permitidos por el bucle:

1. pasar por alto totalmente el bucle
2. pasar una sola vez por el bucle
3. pasar dos veces por el bucle
4. hacer m pasos por el bucle con m < n
5. hacer n -1 y n+l pasos por el bucle



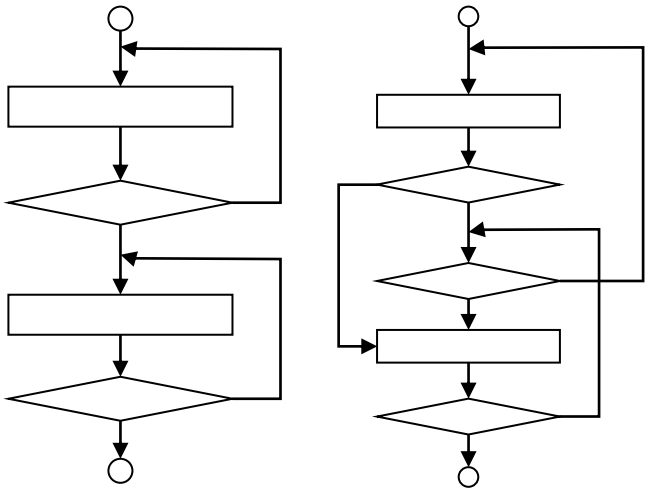
Bucles anidados. Si extendiéramos el enfoque de prueba de los bucles simples a los bucles anidados, el número de posibles pruebas aumentaría geométricamente a medida que aumenta el nivel de anidamiento. Esto llevaría a un número impracticable de pruebas. Beizer [BEI9O] sugiere un enfoque que ayuda a reducir el número de pruebas:



1. Comenzar por el bucle más interior. Establecer o configurar los demás bucles con sus valores mínimos.
2. Llevar a cabo las pruebas de bucles simples para el bucle más interior mientras se mantienen los parámetros de iteración (por ejemplo, contador del bucle) de los bucles externos en sus valores mínimos. Añadir otras pruebas para valores fuera de rango o excluidos.
3. Progresar hacia fuera, llevando a cabo pruebas para el siguiente bucle, pero manteniendo todos los bucles externos en sus valores mínimos y los demás bucles anidados en sus valores «típicos».
4. Continuar hasta que se hayan probado todos los bucles.

Bucles concatenados. Los bucles concatenados se pueden probar mediante el enfoque anteriormente definido para los bucles simples, mientras cada uno de los bucles sea independiente del resto. Sin embargo, si hay dos bucles concatenados y se usa el controlador del bucle 1 como valor inicial del bucle 2, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda usar el enfoque aplicado para los bucles anidados.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles se deben *rediseñar* para que se ajusten a las construcciones de programación estructurada



Bucles Concatenados Bucles no estructurados

6.5.1.- Utilización de la complejidad ciclomática de McCabe

La utilización de la métrica de McCabe ha sido muy popular en el diseño de pruebas desde su creación. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la

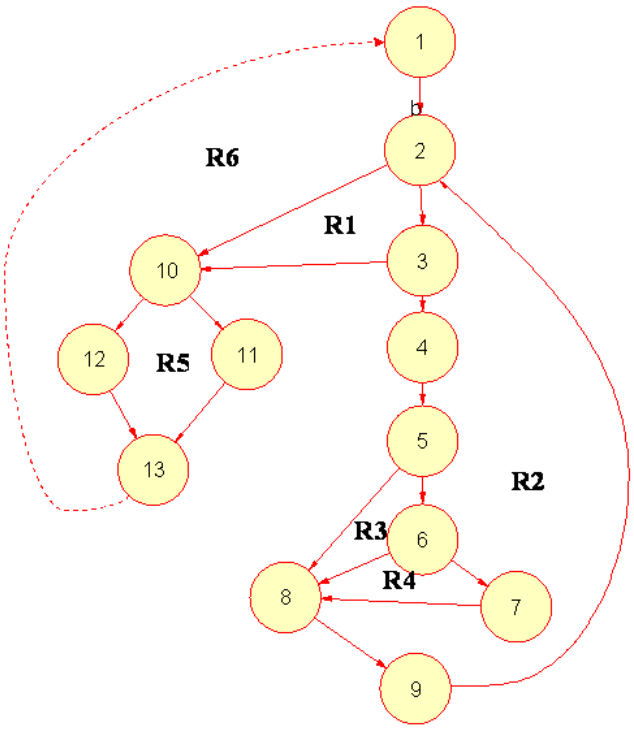
consecución de la ejecución de un conjunto de caminos independientes, lo que implica probar un número de caminos igual al de la métrica. Se propone este criterio como equivalente a una cobertura de decisiones, aunque se han propuesto contraejemplos que invalidan esta suposición.

McCabe habla de caminos linealmente independientes como una idea basada en los conceptos matemáticos de base de vectores y vectores linealmente independientes. En la práctica, puede decirse que un camino es independiente de otros si incorpora un arco o arista del grafo que los demás no incluyen, aunque ésta no es una equivalencia rigurosa. La métrica de McCabe coincide con el número máximo de caminos independientes que puede haber en un grafo.

La complejidad de McCabe V(G) se puede calcular de las tres maneras siguientes a partir de un grafo de flujo G:

1. V(G) = *a*-*n*+2. siendo *a* el número de arcos o aristas del grafo y *n* número de nodos.
2. V(G) = *r*, siendo *r* el número de regiones cerradas del grafo.
3. V(G) = *c* + 1, siendo *c* el número de nodos de condición.

Veamos cómo se aplican estas fórmulas sobre el grafo de flujo de la figura*:*



1. V(G) = *a*-*n*+2. = *17-13+2 = 6.* Para facilitar esta tarea podemos marcar los arcos y los nodos con un número, en nuestro ejemplo sólo hemos identificado los nodos.

2. V(G) = 6. Las regiones o áreas cerradas (limitadas por aristas) del grafo son seis. Las hemos marcado en el grafo con **Rn**. Como puede verse, se ha

marcado un área región 6 añadiendo un arco ficticio, discontinuo, desde el nodo 13, nodo de salida, al nodo 1. Esto se debe a que las fórmulas de McCabe sólo son aplicables a grafos fuertemente conexos, es decir, aquellos para los cuales existe un camino entre cualesquiera dos nodos que se elijan. Los programas, con un nodo de inicio y otro de final. no cumplen ésta condición. Por eso, debemos marcar dicho arco o. como alternativa. contabilizar la región externa al grafo como una más.

3. V(G) = *P+1=5+1=6.* Los nodos de condición son el 2, el 3, el 5, el 6 y el 10. Todos ellos son nodos de decisión binaria, es decir, surgen dos aristas de ellos. En el caso de que de un nodo de condición (por ejemplo, una sentencia Case-of) partieran n arcos (n>2), debería contabilizarse como n-1 para la fórmula (que equivale al número de bifurcaciones binarias necesarias para simular dicha bifurcación «n-aria».

Una vez calculado el valor y V(G) podemos afirmar que el número máximo de caminos independientes de dicho grafo es seis. El criterio de prueba de McCabe consiste en elegir seis caminos que sean independientes entre sí y crear casos de prueba cuya ejecución siga dichos caminos. Para ayudar a la elección de dichos caminos, McCabe creó un procedimiento llamado «método del camino básico», consistente en realizar variaciones sobre la elección de un primer camino de prueba típico denominado camino básico.

Los caminos básicos seleccionados en nuestro caso, descritos como secuencias de nodos, serían los siguientes:

1. 1-2-10-11-13
2. 1-2-10-12-13
3. 1-2-3-10-11-13
4. 1-2-3-4-5-8-9-2-10-11-13
5. 1-2-3-4-5-6-8-9-2-10-11-13
6. 1-2-3-4-5-6-7-8-9-2-10-11-13

Hemos subrayado los elementos de cada camino que lo hacen independiente de los demás. Conviene aclarar que algunos de los caminos quizás no se puedan ejecutar solos y requieran una concatenación con algún otro. A partir de estos caminos, el diseñador de las pruebas debe analizar el código para saber los datos de entrada necesarios para forzar la ejecución de cada uno de ellos. Una vez determinados los datos de entrada hay que consultar la especificación para averiguar cuál es la salida teóricamente correcta para cada caso.

Puede ocurrir también que las condiciones necesarias para que la ejecución pase por un determinado camino no se puedan satisfacer de ninguna manera. Nos encontraríamos entonces ante un «camino imposible». En ese caso, debemos sustituir dicho camino por otro posible que permita satisfacer igualmente el criterio de prueba de McCabe, es decir, que ejecute la misma arista o flecha que diferencia al imposible de los demás caminos independientes.

La experimentación con la métrica de McCabe ha dado como resultado las siguientes conclusiones:

- V(G) marca un límite mínimo de número de casos de prueba para un programa, contando siempre cada condición de decisión como un nodo individual.
- Parece que cuando V(G) es mayor que diez la probabilidad de defectos en el módulo o en el programa crece bastante si dicho valor alto no se debe a sentencias Case-of o similares. En estos casos, es recomendable replantearse el diseño modular obtenido, dividiendo los módulos para no superar el límite de diez de la métrica de McCabe en cada uno de ellos.

6.6.- Prueba Funcional

La prueba funcional o de caja negra se centra en el estudio de la especificación del software, del análisis de las funciones que debe realizar y de las entradas y de las salidas. Lamentablemente, la prueba exhaustiva de caja negra también es generalmente impracticable: pensemos en el ejemplo de la suma visto anteriormente. De nuevo, ya que no podemos ejecutar todas las posibilidades de funcionamiento y todas las combinaciones de entradas y de salidas, debemos buscar criterios que permitan elegir un subconjunto de casos cuya ejecución aporte una cierta confianza en detectar los posibles defectos del software. Para fijar estas pautas de diseño de pruebas, nos apoyaremos en las siguientes dos definiciones de Myers que definen qué es un caso de prueba bien elegido:

- El que reduce el número de otros casos necesarios para que la prueba sea razonable. Esto implica que el caso ejecute el máximo número de posibilidades de entrada diferentes para así reducir el total de casos.
- Cubre un conjunto extenso de otros casos posibles, es decir, nos indica algo acerca de la ausencia o la presencia de defectos en el conjunto específico de entradas que prueba, así como de otros conjuntos similares.

Veremos a continuación distintas técnicas de diseño de casos de caja negra.

6.6.1.- Particiones o clases de equivalencia

Esta técnica utiliza las cualidades que definen un buen caso de prueba de la siguiente manera:

- Cada caso debe cubrir el máximo número de entradas.
- Debe tratarse el dominio de valores de entrada dividido en un número finito de clases de equivalencia que cumplan la siguiente propiedad: la prueba de un valor representativo de una clase permite suponer «razonablemente» que el resultado obtenido (existan defectos o no) será el mismo que el obtenido probando cualquier otro valor de la clase.

El método de diseño de casos consiste entonces en:

- Identificación de clases de equivalencia.
- Creación de los casos de prueba correspondientes.

Para identificar las posibles clases de equivalencia de un programa a partir de su especificación se deben seguir los siguientes pasos:

1. Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada.
2. A parir de ellas, se identifican clases de equivalencia que pueden ser:

- De datos válidos.
- De datos no válidos o erróneos.

La identificación de las clases se realiza basándose en el principio de igualdad de tratamiento: todos los valores de la clase deben ser tratados de la misma manera por el programa.

3. Existen algunas reglas que ayudan a identificar clases:

R1.- Si se especifica un rango de valores para los datos de entrada (por ejemplo. «el número estará comprendido entre 1 y 49»), se creará una clase válida ($1 \leq$ número ≤ 49) y dos clases no válidas (número< l y número>49).

R2.- Si se especifica un número de valores (por ejemplo. «se pueden registrar de uno a tres propietarios de un piso»), se creará una clase válida ($1 \leq$ propietarios ≤ 3) y dos no válidas (propietarios < l y propietarios > 3).

R3.- Si se especifica una situación del tipo «debe ser» o booleana (por ejemplo, «el primer carácter debe ser una letra»), se identifican una clase válida («es una letra») y una no válida («no es una letra»).

R4.- Si se especifica un conjunto de valores admitidos (por ejemplo, «pueden registrarse tres tipos de inmuebles: pisos, chalés y locales comerciales») Y se sabe que el programa trata de forma diferente cada uno de ellos, se identifica una clase válida por cada valor (en este caso son tres: piso, chalé y local) y una no válida (cualquier otro caso: por ejemplo, plaza de garaje).

R5.- En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

La aplicación de estas reglas para la derivación de clases de equivalencia permite desarrollar los casos de prueba para cada elemento de datos del dominio de entrada. La división en clases deberían realizarla personas independientes al proceso de desarrollo del programa, ya que. si lo hace la persona que preparó la especificación o diseño el software, la existencia de algunas clases (en concreto, las no consideradas en el tratamiento) no será, probablemente, reconocida.

El último paso del método es el uso de as clases de equivalencia para identificar los casos de prueba correspondientes. Este proceso consta de las siguientes fases:

1. Asignación de un número único a cada clase de equivalencia.

2. Hasta que todas las clases de equivalencia hayan sido cubiertas por (incorporadas a) casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para cada una de las clases no válidas sin cubrir.

La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares, Por ejemplo, en un programa donde hay que «introducir cantidad (1-99) y letra inicial (A-Z)» ante el caso «105 &» (dos errores), se puede indicar sólo el mensaje «105 fuera de rango de cantidad» y dejar sin examinar el resto de la entrada (el error de introducir «&» en vez de una letra). En otro caso, más frecuente, el programa respondería con un ambiguo «dato incorrecto, introduzca valor» que podría fácilmente hacer creer al equipo de pruebas que los errores fueron correctamente identificados cuando sólo se analizó uno.

Veamos un ejemplo de aplicación de la técnica. Se trata de una aplicación bancaria en la que el operador deberá proporcionar un código, un nombre para que el usuario identifique la operación (por ejemplo, «nómina») y una orden que disparará una serie de funciones bancarias.

Especificación

1. Código área: número de 3 dígitos que no empieza por 0 ni por 1.
2. Nombre de identificación: 6 caracteres.
3. Órdenes posibles: «cheque», «depósito», «pago factura», «retirada de fondos».

Aplicación de las reglas

1. Código
   - número, regla 3, booleana:
     - clase válida (número)
     - clase no válida (no es número)
   - regla 5. la clase número debe subdividirse; por la regla 1, rango, obtenemos:
     - subclase válida (200 código 999)
     - dos subclases no válidas (código < 200; código > 999**)**
2. Nombre de id.. número específico de valores, regla 2:
   - clase válida (6 caracteres)
   - dos clases no válidas (más de 6; menos de 6 caracteres)
3. Orden, conjunto de valores, regla 4:
   - una clase válida para cada orden (“cheque”, “depósito”...); 4 en total.
   - una clase no válida “divisas”, por ejemplo).

En la tabla siguiente se han enumerado las clases identificadas y la generación de casos (presuponiendo que el orden de entrada es código-nombre-orden) se ofrece a Continuación.

Condición de entrada	Clases válidas	Clases inválidas
Código área	(1) 200_código_999	(2) código <200

		(3) código >999 (4) no es número
Nombre para identificar la operación	(5) seis caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) «cheque» (9) «depósito» (l0) «pago factura» (11) «retirada fondos»	(12) ninguna orden válida

Tabla de clases de equivalencia del ejemplo

Casos válidos:

- 300 Nómina Depósito (1) (5) (9)
- 400 Viajes Cheque (1) (5) (8)
- 500 Coches Pago-factura (1) (5) (10)
- 600 Comida Retirada-fondos (1) (5) (11)

Casos no válidos;

- 180 Viajes Pago-factura (2) (5) (10)
- 1032 Nómina Depósito (3) (5) (9)
- XY Compra Retirada-fondos (4) (5) (11)
- 350 A Depósito (1) (6) (9)
- 450 Regalos Cheque (1) (7) (8)
- 550 Casa &%4 (1) (5) (12)

6.6.2.- Análisis de Valores Limite (AVL)

Mediante a experiencia (e incluso a través de demostraciones) se ha podido constatar que los casos de prueba que exploran las condiciones límite de un programa producen un mejor resultado para la detección de defectos, es decir, es más probable que los defectos del software se acumulen en estas condiciones. Podemos definir las condiciones límite como las situaciones que se hallan directamente arriba, abajo y en los márgenes de las clases de equivalencia.

El análisis de valores límite es un técnica de diseño de casos que complementa a la de particiones de equivalencia. Las diferencias entre ambas son las siguientes:

- Más que elegir «cualquier» elemento como representativo de una clase de equivalencia. se requiere la selección de uno o más elementos tal que os márgenes se sometan a prueba.

- Más que concentrarse únicamente en el dominio de entrada (condiciones de entrada) los casos de prueba se generan considerando también el espacio de salida.

El proceso de selección de casos es también heurístico, aunque existen ciertas reglas orientativas. Aunque parezca que el AVL es simple de usar (a la vista de tas reglas), su

aplicación tiene múltiples matices que requieren un gran cuidado a la hora de diseñar las pruebas. Las reglas para identificar clases son las siguientes:

1. Si una condición de entrada especifica un rango de valores («-1.0 valor +1.0») se deben generar casos para los extremos del rango (-1.0 y +1.0) y casos no válidos para situaciones justo más allá de los extremos (-1001 y +1.001, en el caso en que se admitan 3 decimales)
2. Si la condición de entrada especifica un número de valores («el fichero de entrada tendrá de 1 a *255* registros»), hay que escribir casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo de valores (0, 1, *255* y 256 registros).
3. Usar la regla 1 para la condición de salida «<el descuento máximo aplicable en compra al contado será el 50%, el mínimo será el 6%»). Se escribirán casos para intentar obtener descuentos de 5,99%, 6%, 50% y 50,01%.
4. Usar la regla 2 para cada condición de salida («el programa puede mostrar de 1 a 4 listados»). Se escriben casos para intentar generar 0, 1,4 y 5 listados.

   En esta regla, como en la 3, debe recordarse que:

   - Los valores límite de entrada no generan necesariamente los valores límite de salida (recuérdese la función seno, por ejemplo).
   - No siempre se pueden generar resultados fuera del rango de salida (pero es interesante considerarlo).
   - Si la entrada o la salida de un programa es un conjunto ordenado (por ejemplo, una tabla. un archivo secuencial, etc.). los casos se deben concentrar en el primero y en el último elemento.

Es recomendable utilizar el ingenio para considerar todos los aspectos y matices, a veces sutiles, en la aplicación del AVL.

6.6.3.- Conjetura de errores

La idea básica de esta técnica consiste en enumerar una lista de equivocaciones que pueden cometer los desarrolladores y de las situaciones propensas a ciertos errores. Después se generan casos de prueba en base a dicha lista (se suelen corresponder con defectos que aparecen comúnmente y no con aspectos funcionales). Esta técnica también se ha denominado generación de casos (o valores) especiales, ya que no se obtienen en base a otros métodos sino mediante la intuición o la experiencia.

No existen directrices eficaces que puedan ayudar a generar este tipo de casos, ya que lo único que se puede hacer es presentar algunos ejemplos típicos que reflejan esta técnica. Algunos valores a tener en cuenta para los casos especiales son los siguientes:

1. El valor cero es una situación propensa a error tanto en la salida como en la entrada.
2. En situaciones en las que se introduce un número variable de valores (por ejemplo. una lista), conviene centrarse en el caso de no introducir ningún valor y en el de un solo valor. También puede ser interesante una lista que tiene todos los valores iguales.

3. Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación.
4. También interesa imaginar lo que el usuario puede introducir como entrada a un programa. Se dice que se debe prever toda clase de acciones de un usuario como si fuera «completamente tonto» o, incluso, como si quisiera sabotear el programa.

6.6 4.- Pruebas aleatorias

En las pruebas aleatorias simulamos la entrada habitual del programa creando datos de entrada en la secuencia y con la frecuencia con las que podrían aparecer en la práctica de forma continua: Esto implica usar una herramienta denominada generador de pruebas, a las que se alimenta con una descripción de las entradas y las secuencias de entrada posibles y su probabilidad de ocurrir en la práctica. Este enfoque de prueba es muy común en la prueba de compiladores en la que se generan aleatoriamente códigos de programas que sirven de casos de prueba para la compilación.

Si el proceso de generación se ha realizado correctamente, se crearán todas las posibles entradas del programa en todas las posibles combinaciones y, permutaciones. También se puede conseguir, indicando la distribución estadística que siguen, que la frecuencia de las entradas sea la apropiada para orientar correctamente nuestras pruebas hacia lo que es probable que suceda en la práctica. No obstante, esta forma de diseñar casos de prueba es menos utilizada que las de caja blanca y de caja negra

6.6.5.- Métodos de prueba basados en grafos

El primer paso en la prueba de caja negra es entender los objetos que se modelan en el software y las relaciones que conectan a estos objetos. Una vez que se ha llevado a cabo esto, el siguiente paso es definir una serie de pruebas que verifiquen que «todos los objetos tienen entre ellos las relaciones esperadas». Dicho de otra manera, la prueba del software empieza creando un grafo de objetos importantes y sus relaciones, y después diseñando una serie de pruebas que cubran el grafo de manera que se ejerciten todos los objetos y sus relaciones para descubrir los errores.

Para llevar a cabo estos pasos, el ingeniero del software empieza creando un *grafo* —una colección de *nodos* que representan objetos: *enlaces* que representan las relaciones entre los objetos: *pesos de nodos* que describen las propiedades de un nodo (por ejemplo, un valor específico de datos o comportamiento de estado> y *pesos de enlaces* que describen alguna característica de un enlace—.

En la Figura l7.9a se muestra una representación simbólica de un grafo. Los nodos se representan como círculos conectados por enlaces que toman diferentes formas. Un *enlace dirigido* (representado por una flecha) indica que una relación se mueve sólo en una dirección.

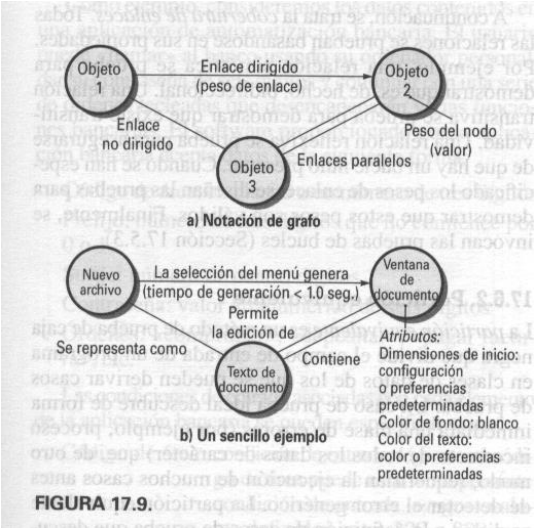


FIGURA 17.9.

Un *enlace bidireccional*, también denominado *enlace simétrico,* implica que la relación se aplica en ambos sentidos. Los enlaces paralelos se usan cuando se establecen diferentes relaciones entre los nodos del grafo.

Como ejemplo sencillo, consideremos una parte de un grafo de una aplicación de un procesador de texto (Fig. 17.9b) donde:

objeto nº 1 = selección en el menú de archivo nuevo

objeto nº 2 = ventana del documento

objeto nº 3 = texto del documento

Como se muestra en la figura, una selección del menú en archivo nuevo genera una ventana del documento. El peso del nodo de ventana del documento proporciona una lista de los atributos de la ventana que se esperan cuando se genera una ventana. El peso del enlace indica que la ventana se tiene que generar en menos de 1.0 segundos. Un enlace no dirigido establece una relación simétrica entre selección en el menú de archivo nuevo y texto del documento, y los enlaces paralelos indican las relaciones entre la ventana del documento y el texto del documento. En realidad, se debería generar un grafo bastante más detallado como precursor al diseño de casos de prueba. El ingeniero del software obtiene entonces casos de prueba atravesando el grafo y cubriendo cada una de las relaciones mostradas. Estos casos de prueba están diseñados para intentar encontrar errores en alguna de las relaciones.

Beizer describe un número de métodos de prueba de comportamiento que pueden hacer uso de los grafos:

Modelado del flujo de transacción. Los nodos representan los pasos de alguna transacción (por ejemplo, los pasos necesarios para una reserva en una línea aérea usando un servicio en línea), y los enlaces representan las conexiones lógicas entre los pasos (por ejemplo, *vuelo.información.entrada* es seguida de *validación/disponibilidad.procesamiento).* El diagrarna de flujo de datos puede usarse para ayudar en la creación de grafos de este tipo.

M*odelado de estado finito.* Los nodos representan diferentes estados del software observables por el usuario (por ejemplo, cada una de las «pantallas» que aparecen

cuando un telefonista coge una petición por teléfono), y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, *petición-información se verifica durante inventario-disponibilidad-búsqueda* y es seguido por *cliente-factura-información- entrada).* El diagrama estado-transición puede usarse para ayudar en la creación de grafos de este tipo.

Modelado del flujo de datos. Los nodos son objetos de datos y los enlaces son las transformaciones que ocurren para convertir un objeto de datos en otro. Por ejemplo, el nodo FICA.impuesto.retenido (FIR) se calcula de brutos.sueldos (BS) usando la relación FIR = 0,62 X BS.

Un estudio detallado de cada uno de estos métodos de prueba basados en grafos se sale de nuestros objetivos. Merece la pena, sin embargo, proporcionar un resumen genérico del enfoque de pruebas basadas en grafos.

Las pruebas basadas en grafos empiezan con la definición de todos los nodos y pesos de nodos. O sea, se identifican los objetos y los atributos. El modelo de datos puede usarse como punto de partida, pero es importante tener en cuenta que muchos nodos pueden ser objetos de programa (no representados explícitamente en el modelo de datos). Para proporcionar una indicación de los puntos de inicio y final del grafo, es útil definir unos nodos de entrada y salida.

Una *vez* que se han identificado los nodos se deberían establecer los enlaces y los pesos de enlaces. En general, conviene nombrar los enlaces, aunque los enlaces que representan el flujo de control entre los objetos de programa no es necesario nombrarlos.

En muchos casos, el modelo de grafo puede tener bucles (por ejemplo, un camino a través del grafo en el que se encuentran uno o más nodos más de una vez). La prueba de bucle se puede aplicar también a nivel de comportamiento (de caja negra). El grafo ayudará a identificar aquellos bucles que hay que probar.

Cada relación es estudiada separadamente, de manera que se puedan obtener casos de prueba. La *transitividad* de relaciones secuenciales es estudiada para determinar cómo se propaga el impacto de las relaciones a través de objetos definidos en el gafo. La transitividad puede ilustrarse considerando tres objetos X, Y y Z. Consideremos las siguientes relaciones:

X *es necesaria para calcular Y*

Y es necesaria para calcular Z

Por tanto. se ha establecido una relación transitiva entre X y Z:

X *es necesaria para calcular* Z

Basándose en esta relación transitiva, las pruebas para encontrar errores en el cálculo de Z deben considerar una variedad de valores para X e Y.

La *simetría* de una relación (enlace de grafo) es también una importante directriz para diseñar casos de prueba. Si un enlace es bidireccional (simétrico), es importante probar esta característica. La característica *UNDO* (deshacer) en muchas aplicaciones

para ordenadores personales implementa una limitada simetría. Es decir, *UNDO* permite deshacer una acción después de haberse completado. Esto debería probarse minuciosamente y todas las excepciones (por ejemplo, lugares donde no se puede usar *UNDO)* deberían apuntarse. Finalmente, todos los nodos del grafo deberían tener una relación que los lleve devuelta a ellos mismos; en esencia un bucle de «no acción» o «acción nula». Estas relaciones *reflexivas* deberían probarse también.

Cuando empieza el diseño de casos de prueba, el primer objetivo es conseguir la *cobertura de nodo.* Esto significa que las pruebas deberían diseñarse para demostrar que ningún nodo se ha omitido inadvertidamente y que los pesos de nodos (atributos de objetos) son correctos.

A continuación, se trata la *cobertura de enlaces.* Todas las relaciones se prueban basándose en sus propiedades. Por ejemplo, una relación simétrica se prueba para demostrar que es, de hecho, bidireccional. Una relación transitiva se prueba para demostrar que existe transitividad. Una relación reflexiva se prueba para asegurarse de que hay un bucle nulo presente. Cuando se han especificado los pesos de enlace, se diseñan las pruebas para demostrar que estos pesos son válidos. Finalmente, se invocan las pruebas de bucles.

6.7.- Enfoque práctico recomendado para el diseño de casos

Los dos enfoques estudiados, caja blanca y caja negra, representan aproximaciones diferentes para las pruebas. El enfoque práctico recomendado para el uso de las técnicas de diseño de casos pretende mostrar el uso más apropiado de cada técnica para la obtención de un conjunto de casos útiles sin perjuicio de las estrategias de niveles de prueba

1. Si la especificación contiene combinaciones de condiciones de entrada, comenzar formando sus grafos de causa-efecto (ayudan a la comprensión de dichas combinaciones).
2. En todos los casos, usar el análisis de valores-límites para añadir casos de prueba: elegir límites para dar valores a las causas en Los casos generados asumiendo que cada causa es una clase de equivalencia.
3. Identificar las clases válidas y no válidas de equivalencia para la entrada y la salida, y añadir los casos no incluidos anteriormente (¿cada causa es una única clase de equivalencia? ¿Deben dividirse en clases?).
4. Utilizar la técnica de conjetura de errores para añadir nuevos casos, referidos a valores especiales.
5. Ejecutar los casos generados hasta el momento (de caja negra) y analizar la cobertura obtenida (en este punto ofrecen una gran ayuda las herramientas de análisis de cobertura.
6. Examinar la lógica del programa para añadir los casos precisos (de caja blanca) para cumplir el criterio de cobertura elegido si los resultados de La ejecución del punto 5 indican que no se ha satisfecho el criterio de cobertura elegido (que figura en el plan de pruebas).

Aunque éste es el enfoque integrado para una prueba «razonable», en la práctica la aplicación de las distintas técnicas está bastante discriminada según la etapa de la

estrategia de prueba.

Otra cuestión importante en relación a los dos tipos de diseño de pruebas es: ¿por qué incluir técnicas de caja blanca para explorar detalles lógicos y no centrarnos mejor en probar los requisitos funcionales con técnicas de caja negra? (si el programa realiza correctamente las funciones ¿por qué hay que intentar probar un determinado número de caminos?).

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa (a menor probabilidad de ejecutarse un camino, mayor número de errores).
- Se suele creer que un determinado camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar regularmente.
- Los errores tipográficos son aleatorios; pueden aparecer en cualquier parte del programa (sea muy usada o no).
- La probabilidad y La importancia de un trozo de código suele ser calculada de forma muy subjetiva.

Debemos recordar que tanto La prueba exhaustiva de caja blanca como de caja negra son impracticables. ¿Bastaría, no obstante, una prueba exhaustiva de caja blanca solamente? Véase el siguiente programa:

```
IF (x+y+z)/3 = x
THEN print («X, Y y Z son iguales»)
ELSE print («X. Y y Z no son iguales»)
```

En este programa, una prueba exhaustiva de caja blanca (que pase por todos los caminos) no asegura necesariamente la detección de los defectos de su diseño. véase. por ejemplo, cómo los dos casos siguientes no detectan ningún problema en el programa:

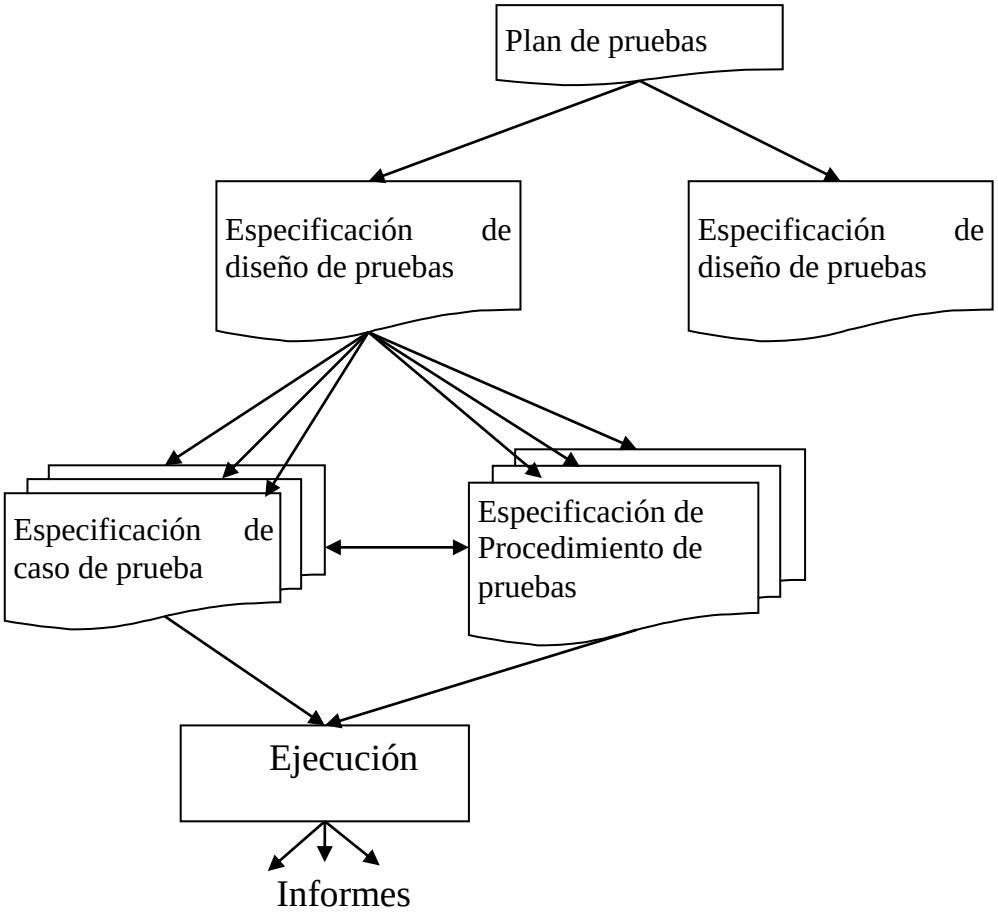
```
x=5, y=5, z=5
x=2, y=3, z=7
```

Sin embargo, el código tiene un problema. Véase el caso x=5, y=3, z=7, o x=4, y=5, z=3

Estos contraejernplos no pretenden influir en el tipo de técnica de diseño de casos que debernos elegir. Más bien nos indican que conviene emplear lo mejor de todas las técnicas para obtener pruebas más eficaces. Podemos citar una frase de B. Beizer a favor de las pruebas de caja blanca: «Los errores se esconden en los rincones y se acumulan en los límites».

6.8.- Documentación del diseño de las pruebas

Recordemos que la documentación de las pruebas es necesaria para una buena organización de las mismas, así como para asegurar su reutilización que es fundamental para optimizar tanto la eficacia como la eficiencia de las pruebas. Los distintos documentos de trabajo de las pruebas, según el estándar IEEE std. 829 son los que se reflejan en la figura.



Los documentos contemplados en el estándar se asocian a las distintas fases de las pruebas de la siguiente manera:

- El primer paso se sitúa en la planificación general del esfuerzo de prueba (plan de pruebas) para cada fase de la estrategia de prueba para el producto
- Se genera inicialmente la especificación del diseño de la prueba (que surge de la ampliación y el detalle del plan de pruebas).
- A partir de este diseño, se pueden definir con detalle cada uno de los casos mencionados escuetamente en el diseño de la prueba (se fijan los datos de prueba exactos, los resultados esperados, etc.).
- Tras generar los casos de prueba detallados, se debe especificar cómo proceder en detalle en la ejecución de dichos casos (procedimientos de prueba).
- Tanto las especificaciones de casos de prueba corno las especificaciones de los procedimientos deben ser los documentos básicos para la ejecución de las pruebas. No obstante, son los procedimientos los que determinan realmente cómo se desarrolla la ejecución.

A continuación, describiremos brevemente los contenidos de los distintos documentos contemplados en el estándar.

6.8.1.- Plan de pruebas

Objetivo del documento: señalar el enfoque, los recursos y el esquema de actividades de prueba, así como los elementos a probar, las características, las actividades de prueba, el personal responsable y los riesgos asociados.

- *Estructura fijada en el estándar:*

1. Identificador único del documento (para la gestión de configuración).
2. Introducción y resumen de elementos y características a probar.
3. Elementos software que se van a probar (por ejemplo, programas o módulos).
4. Características que se van a probar.
5. Características que no se prueban.
6. Enfoque general de la prueba (actividades, técnicas, herramientas, etc.).
7. Criterios de paso/fallo para cada elemento.
8. Criterios de suspensión y requisitos de reanudación.
9. Documentos a entregar (como mínimo, los descritos en el estándar).
10. Actividades de preparación y ejecución de pruebas.
11. Necesidades de entorno.
12. Responsabilidades en la organización y realización de las pruebas.
13. Necesidades de personal y de formación.
14. Esquema de tiempos (con tiempos estimados, hitos, etc.).
15. Riesgos asumidos por el plan y planes de contingencias para cada riesgo.
16. Aprobaciones y firmas con nombre y puesto desempeñado.

6.8.2.- Especificación del diseño de pruebas

Objetivo del documento: especificar los refinamientos necesarios sobre el enfoque general reflejado en el plan e identificar las características que se deben probar con este diseño de pruebas.

Estructura fijada en el estándar:

1. Identificador (único) para la especificación Proporcionar también una referencia del plan asociado (si existe).
2. Características a probar de los elementos software (y combinaciones de características).
3. Detalles sobre el plan de pruebas del que surge este diseño, incluyendo las técnicas de prueba específica y los métodos de análisis de resultados.
4. Identificación de cada prueba:
   - Identificador.
   - Casos que se van a utilizar.
   - Procedimientos que se van a seguir
5. Criterios de paso/fallo de la prueba (criterios para determinar si una característica o combinación de características ha pasado con éxito la prueba o no).

6.8.3.- Especificación de caso de prueba

Objetivo del documento: definir uno de los casos de prueba identificado por una especificación del diseño de las pruebas.

Estructura fijada en el estándar:

1. Identificador único de la especificación.
2. Elementos software (por ejemplo, módulos que se van a probar: definir dichos elementos y las características que ejercitará este caso
3. Especificaciones de cada entrada requerida para ejecutar el caso (incluyendo las relaciones entre las diversas entradas; por ejemplo, la sincronización de las mismas).
4. Especificaciones de todas las salidas y las características requeridas (por ejemplo, el tiempo de respuesta) para los elementos que se van a probar.
5. Necesidades de entorno (hardware, software y otras como, por ejemplo, el personal).
6. Requisitos especiales de procedimiento (o restricciones especiales en los procedimientos para ejecutar este caso).
7. Dependencias entre casos (por ejemplo, listar los identificadores de los casos que se van a ejecutar antes de este caso de prueba).

6.8.4.- Especificación de procedimiento de prueba

Objetivo del documento: especificar los pasos para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados pasa analizar un elemento software con el propósito de evaluar un conjunto de características del mismo.

Estructura fijada en el estándar:

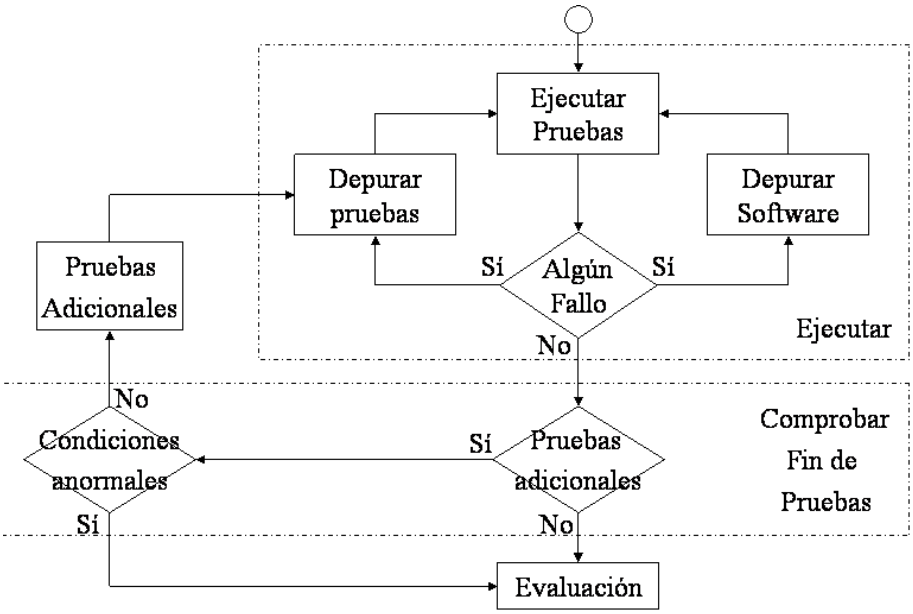
1. Identificador único de la especificación y referencia a la correspondiente especificación de diseño de prueba.
2. Objetivo del procedimiento y lista de casos que se ejecutan con él.
3. Requisitos especiales para la ejecución (por ejemplo, entorno especial o personal especial).
4. Pasos en el procedimiento. Además de la manera de registrar los resultados y los incidentes de la ejecución, se debe especificar:
   - La secuencia necesaria de acciones para preparar la ejecución
   - Acciones necesarias para empezar la ejecución.
   - Acciones necesarias durante la ejecución.
   - Cómo se realizarán las medidas (por ejemplo, el tiempo de respuesta>.
   - Acciones necesarias para suspender la prueba (cuando los acontecimientos no previstos lo obliguen).
   - Puntos para reinicio de la ejecución y acciones necesarias para el reinicio en estos puntos.
   - Acciones necesarias para detener ordenadamente la ejecución.

- Acciones necesarias para restaurar el entorno y dejarlo en la situación existente antes de as pruebas.
- Acciones necesarias para tratar los acontecimientos anómalos

6.9.- Ejecución de las pruebas

6.9.1.- El proceso de ejecución

El proceso de las pruebas según el estándar IEEE std. 1008 en el que se incluye la ejecución de pruebas es el mostrado en la figura.



El proceso abarca las siguientes fases:

- Ejecutar las pruebas, cuyos casos y procedimientos han sido ya diseñados previamente.
- Comprobar si se ha concluido el proceso de prueba (según ciertos criterios de compleción de prueba que suelen especificarse en e1 plan de pruebas)
- En el caso de que hayan terminado las pruebas. se evalúan los resultados: en caso contrario, hay que generar las pruebas adicionales para que se satisfagan los criterios de compleción de pruebas.

El proceso de ejecución propiamente dicho consta de las siguientes fases:

- Se ejecutan las pruebas, es decir, se realiza el paso por el ordenador de los datos de prueba
- Se comprueba si ha habido algún fallo al ejecutar (por ejemplo, se cae el sistema, se bloquea el teclado, etc., es decir, cualquier hecho que impide terminar la ejecución de algún caso).
- Si lo ha habido, se puede deber a un defecto software, lo que nos lleva al proceso de depuración o corrección del código. Se puede deber también a un

defecto en el propio diseño de las pruebas (por ejemplo, resulta imposible al sistema operativo, no al programa que se prueba, adaptarse a las condiciones o entorno que exigen las pruebas). En ambos casos, las nuevas pruebas o las corregidas se deberán ejecutar en el ordenador.

- De no existir fallo, se pasará a la comprobación de la terminación de las pruebas

En la figura se muestra el proceso de comprobación de la terminación de las pruebas. Se pueden observar las siguientes fases:

- Tras la ejecución, se comprobará si se cumplen los criterios de compleción de pruebas descritos en el correspondiente plan de pruebas (por ejemplo, se terminan las pruebas cuando se han probado todos los procedimientos de operación o se ha cumplido la cobertura lógica marcada).
- En caso de terminar las pruebas, se pasa a la evaluación de los productos probados sobre la base de los resultados obtenidos (terminación normal).
- En caso de no terminar las pruebas, se debe comprobar la presencia de condiciones anormales en la prueba (por ejemplo, un defecto importante no corregido ya detectado previamente, tiempo finalizado, etc.).
- Si hubiesen existido condiciones anormales (por ejemplo, no se han podido ejecutar todos los casos porque el sistema se cae regularmente), se pasa de nuevo a la evaluación (terminación anormal); en caso contrario se pasa a generar y ejecutar pruebas adicionales para satisfacer cualquiera de las dos terminaciones.

Los criterios de compleción de pruebas hacen referencia a las áreas (características, funciones, instrucciones, etc.) que deben cubrir las pruebas y el grado de cobertura para cada una de esas áreas. Como ejemplos genéricos de este tipo de criterios se pueden indicar los siguientes:

- Se debe cubrir cada característica del software mediante un caso de prueba o una excepción aprobada en el plan de pruebas.
- En programas codificados con lenguajes procedimentales. se deben cubrir todas las instrucciones ejecutables mediante casos de prueba (o se deben marcar as posibles excepciones aprobadas en el plan de pruebas).

6.9.2- Documentación de la ejecución de pruebas

Al igual que en el diseño de las pruebas, la documentación de la ejecución de las pruebas es fundamental para la eficacia en la detección y corrección de defectos, así como para dejar constancia de los resultados de las pruebas.

La figura siguiente nos muestra el conjunto de documentos que se relaciona directamente con la ejecución de las pruebas según el estándar IEEE std. 829. Se pueden distinguir dos grupos principales:

1. Documentación de entrada, constituida principalmente por las especificaciones de los casos de prueba que se van a usar y las especificaciones de los procedimientos de pruebas (ambas ya explicadas con anterioridad).

2. Documentación de salida o informes sobre la ejecución. Cada ejecución de pruebas generará dos tipos de documentos
   - Histórico de pruebas o registro cronológico de la ejecución.
   - Informes de los incidentes ocurridos (si hay) durante la ejecución.
3. La documentación de salida correspondiente a un mismo diseño de prueba se recoge en un informe resumen de pruebas.

6.9.3.- Histórico de pruebas

Objetivo: el histórico de pruebas (*test log*) documenta todos los hechos relevantes ocurridos durante la ejecución de las pruebas.

Estructura fijada en el estándar:

1. Identificador.
2. Descripción de la prueba: elementos probados y entorno de la prueba
3. Anotaciones de datos sobre cada hecho ocurrido (incluido el comienzo y el final de la prueba):
   - Fecha y hora.
   - Identificador de informe de incidente.
4. Otras informaciones

6.9.4.- Informe de incidente

Objetivo: el informe de incidente *(test incident report)* documenta cada incidente (por ejemplo, una interrupción en las pruebas debido a un corte de electricidad, bloqueo del teclado, etc.) ocurrido en la prueba y que requiera una posterior investigación.

Estructura fijada en el estándar:

1. Identificador.
2. Resumen del incidente.
3. Descripción de datos objetivos (fecha/hora, entradas, resultados esperados, etc.).
4. Impacto que tendrá sobre las pruebas.

6.9.5.- Informe resumen de las pruebas

Objetivo: el informe resumen (test *summary report)* resume los resultados de las actividades de prueba (las reseñadas en el propio informe) y aporta una evaluación del software basada en dichos resultados.

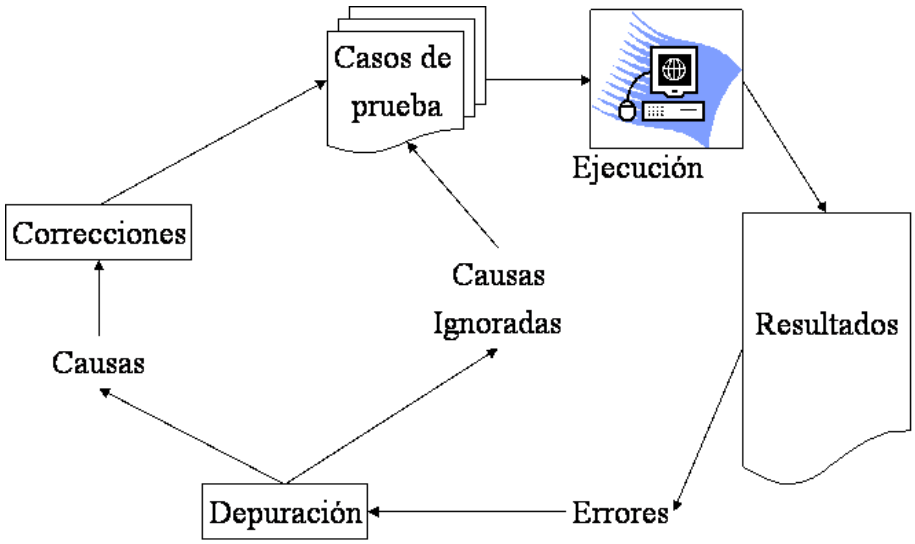
Estructura fijada en el estándar:

1. Identificador.
2. Resumen de la evaluación de los elementos probados.
3. Variaciones del software respecto a su especificación de diseño, así como las variaciones en las pruebas.
4. Valoración de la extensión de la prueba (cobertura lógica, funcional, de requisitos, etc.).
5. Resumen de los resultados obtenidos en las pruebas.

6. Evaluación de cada elemento software sometido a prueba (evaluación general del software incluyendo las limitaciones del mismo).
7. Resumen de las actividades de prueba (incluyendo el consumo de todo tipo de recursos).
8. Firmas y aprobaciones de quienes deban supervisar el informe.

6.9.6.- Depuración

Se define la depuración como «el proceso de localizar, analizar y corregir los defectos que se sospecha que contiene el software». Suele ser la consecuencia de una prueba con éxito (es decir, que descubre los síntomas de un defecto).



Las consecuencias de la depuración pueden ser dos (véase la figura):

- Encontrar la causa del error, analizarla y corregirla.
- No encontrar la causa y, por lo tanto, tener que generar nuevos casos de prueba que puedan proporcionar información adicional para su localización (casos de prueba para depuración).

Las dos principales etapas en la depuración son las siguientes:

- Localización del defecto, que conlleva la mayor parte del esfuerzo.
- Corrección del defecto, efectuando las modificaciones necesarias en el software.

El proceso de prueba (véase figura 12.13) implica generar unos casos de prueba, ejecutarlos en el ordenador y obtener unos resultados. Dichos resultados se analizan para la búsqueda de síntomas de defectos (errores) en el software. Esta información se pasa al proceso de depuración para obtener las causas del error (defecto). En caso de conseguirlo. se corrige el defecto: en caso contrario. se llevarán a cabo nuevas pruebas que ayuden a localizarlo (reduciendo en cada pasada el posible dominio de existencia del defecto). Tras corregir el defecto, se efectuarán nuevas pruebas que comprueben si se ha eliminado dicho problema.

6.9.6.1.- Consejos para la depuración

G. J. Myers [MYERS. 1979] aporté la siguiente lista de consejos para el proceso de depuración:

- *Localización del error*

  1. **Analizar la información y pensar**. La depuración es un proceso mental de resolución de un problema y lo mejor para el mismo es el análisis. No se debe utilizar un enfoque aleatorio en la búsqueda del defecto.
  2. **Al llegar a un punto muerto, pasar a otra cosa**. Si tras un tiempo razonable no se consiguen resultados, merece la pena refrescar la mente (para empezar de nuevo o para que el inconsciente nos proporcione la solución.
  3. **Al llegar a un punto muerto, describir el problema a otra persona**. El simple hecho de describir a otro el problema nos descubre muchas cosas («cuando todo falle, pedir ayuda»).
  4. **Usar herramientas de depuración sólo como recurso secundario**. Deben ayudar al análisis mental, no pueden pretender sustituirlo (por lo menos, en la actualidad).
  5. **No experimentar cambiando el programa**. Evitar depurar con esta actitud inadecuada que se puede resumir como: «No sé qué está mal, así que cambiará este bucle y veré qué pasa».
  6. **Se deben atacar los errores individualmente**, de uno en uno, so pena de dificultar aún más la depuración
  7. **Se debe fijar la atención también en los datos** manejados en el programa y no sólo en la lógica del proceso.

- *Corrección del error*

  1. **Donde hay un defecto, suele haber más**. (Principio de Pareto) Es una conclusión obtenida de la experiencia. Cuando se corrige un defecto se debe examinar su proximidad inmediata buscando elementos sospechosos.
  2. **Debe corregirse el defecto, no sus síntomas**. Lo que debe corregirse y desaparecer es el defecto, no se trata de intentar enmascarar sus síntomas.
  3. **La probabilidad de corregir perfectamente un defecto no es del 100%.** Por lo tanto, se deben revisar las correcciones antes de implantarlas (mediante técnicas de revisión: walkthroughs, inspecciones, revisiones, etc., antes de la corrección). Después de la corrección, se utilizan pruebas específicas.
  4. **Cuidado con crear nuevos defectos**. Es frecuente crear nuevos defectos al corregir sin cautela. La probabilidad de que un cambio se realice correctamente es del 50% (aproximadamente) según algunos estudios.
  5. **La corrección debe situarnos temporalmente en la fase de diseño**. Hay que mentalizarse de que se reinicia el diseño de la sección de código defectuoso y no sólo se retoca el código.
  6. **Cambiar el código fuente, no el código objeto**. Cambiar el código objeto provoca:

     - experimentación indeseable.
     - falta de sincronización fuente-objeto.

6.9.7.- Análisis de errores a análisis causal

El objetivo del análisis causal es proporcionar información sobre la naturaleza de los defectos (de los cuales sabemos muy poco). Para ello, es fundamental que se recoja la siguiente información de cada defecto detectado:

1. ¿Cuándo se cometió?
2. ¿Quién lo hizo?
3. ¿Qué se hizo mal?
4. ¿Cómo se podría haber prevenido?
5. ¿Por qué no se detectó antes?
6. ¿Cómo se podría haber detectado antes?
7. ¿Cómo se encontró el error?

El objetivo primordial es la formación del personal sobre los errores que cornete para que se puedan prevenir en futuro. Nunca debe usarse esta información para evaluar al personal o para «buscar a quién despedir». El mantenimiento y almacenamiento de un archivo de los defectos detectados y corregidos, anotando dónde ocurrieron los errores y los tipos de defectos encontrados, se puede utilizar para la predicción de los futuros fallos del software. Esta predicción se basa en complejos modelos de fiabilidad que requieren unos conocimientos matemáticos considerables.

6.10.- Estrategia de aplicación de las pruebas

Una vez conocidas las técnicas de diseño y ejecución, debemos analizar cómo se plantea la utilización de las pruebas en el ciclo de vida. La estrategia de aplicación y la planificación de las pruebas pretenden integrar el diseño de los casos de prueba en una serie de pasos bien coordinados a través de la creación de distintos niveles de prueba con diferentes objetivos. Además, permite la coordinación del personal de desarrollo, del departamento de aseguramiento de calidad (o algún grupo independiente dedicado a la V y V o a las pruebas) y del cliente, gracias a la definición de los papeles que deben desempeñar cada uno de ellos y la forma de llevarlos a cabo.

En general, la estrategia de pruebas suele seguir las siguientes etapas:

- Las pruebas comienzan a nivel de módulo.
- Una vez terminadas, progresan hacia la integración del sistema completo y su instalación.
- Culminan cuando el cliente acepta el producto y se pasa a su explotación inmediata.

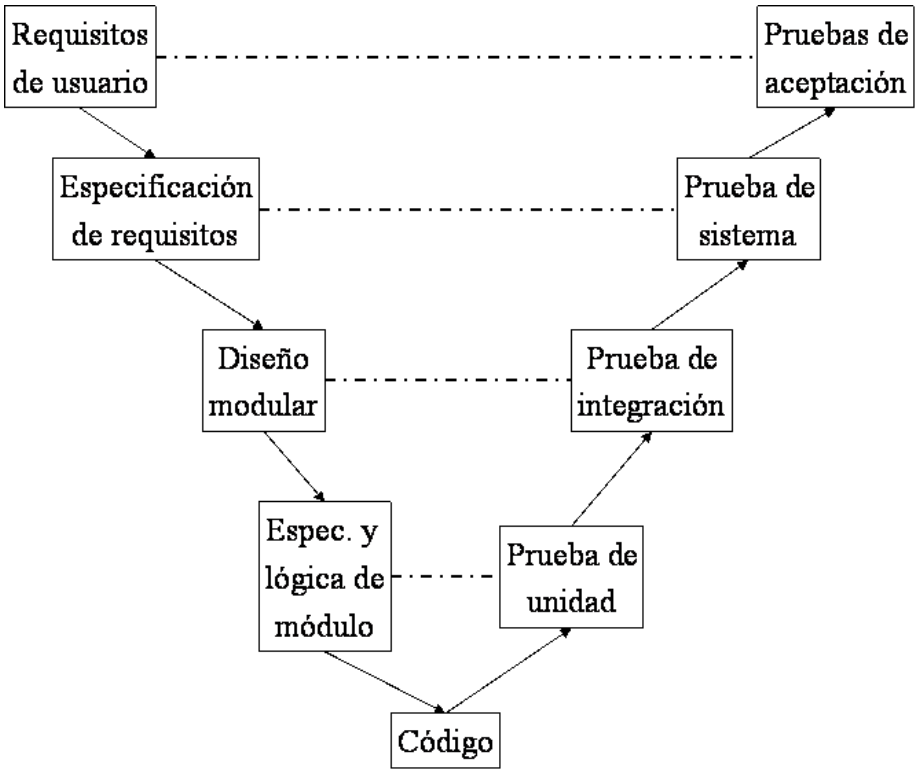
Esta serie típica de etapas se describe con mayor detalle a continuación:

1. Se comienza en la prueba de cada módulo, que normalmente la realiza el propio personal de desarrollo en su entorno.
2. Con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto (prueba de integración).
3. El software totalmente ensamblado se prueba como un conjunto para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimientos, seguridad, etc. (prueba funcional o de validación). Este nivel

de prueba coincide con el de la prueba del sistema cuando no se trate de software empotrado u otros tipos de especiales de aplicaciones.

4. El software ya validado se integra con el resto del sistema (por ejemplo, elementos mecánicos interfaces electrónicas, etc.) para probar su funcionamiento conjunto (prueba del sistema).
5. Por último, el producto final se pasa a la prueba de aceptación para que el usuario compruebe en su propio entorno de explotación si lo acepta como está o no (prueba de aceptación)

Lo cierto es que cada nivel de prueba se centra en probar el software en referencia al trabajo realizado en una diferente etapa de desarrollo. Así (véase la figura dicha relación se concreta en un modelo de ciclo de vida denominado ciclo en uve.



1. La prueba de módulo (prueba de unidad) centra sus actividades en ejercitar la lógica del módulo (caja blanca) y los distintos aspectos de la especificación de las funciones que debe realizar el módulo (caja negra).
2. La prueba de interacción debe tener en cuenta los mecanismos de agrupación de módulos fijados en la estructura del programa, así como, en general, las interfaces entre componentes de la arquitectura del software.
3. La prueba de validación (o funcional) debe comprobar si existen desajustes entre el software y los requisitos fijados para su funcionamiento en la ERS.
4. La prueba del sistema debe centrar sus comprobaciones en el cumplimiento de los objetivos indicados para el sistema.
5. La prueba de aceptación sirve para que el usuario pueda verificar si el producto final se ajusta a los requisitos fijados por él (normalmente en forma

de criterios de aceptación en el contrato) o, en último caso, en función de lo que indique el contrato.

6.11.- Pruebas en desarrollos orientados a objetos

Las técnicas y estrategias presentadas en este capítulo están inspiradas en la aplicación a desarrollos estructurados. Cuando trabajamos con tecnología orientada a objetos, algunas de las técnicas presentadas pierden su interés o deben adaptarse para seguir resultando útiles. Desde el punto de vista del diseño de casos de pruebas:

- Las técnicas de caja negra tienen vigencia toda vez que su uso se centra en el comportamiento de la aplicación. En el caso de pruebas de sistema, son las descripciones de casos de uso las que pueden aportar la referencia sobre la cual diseñar los casos de prueba. De hecho, los escenarios de casos de Uso permiten definir los casos de prueba aplicando técnicas de caja negra sobre los datos y los eventos incluidos en los escenarios. Éstos, a su vez, se pueden identificar en función de los caminos que se pueden trazar en los diagramas de actividad o en los diagramas de estados descriptivos del caso, sin olvidar todos los flujos alternativos o tratamientos de error y excepciones. Evidentemente, la aplicación de valores límite, tratamiento de combinaciones de entrada y conjetura de errores es perfectamente aplicable al software orientado a objetos.

- Las técnicas de caja blanca disminuyen sus posibilidades de aplicación, ya que quedan confinadas a las instrucciones de cada método de cada clase para conseguir el nivel de cobertura más apropiado. Evidentemente la cobertura de sentencias resulta esencial y el tratamiento de bucles y decisiones a través de las coberturas de condiciones y decisiones es más que recomendable. También cabría la posibilidad de aplicar la cobertura de caminos (incluido el criterio de McCabe) no sólo al código de método sino a diagramas de comportamiento con estructura de red: diagramas de estados, de actividad, etc.

En el caso del diseño de pruebas cabe señalar la posibilidad de distinguir criterios según el nivel de pruebas en el que nos situemos. Así, para las pruebas de unidad, es decir, pruebas de clases, la prueba de clases de equivalencia y valores límite es apropiada aplicándola a las entradas y salidas representadas como parámetros de métodos fundamentalmente. Un enfoque basado en clasificar operaciones según el diagrama de estados de la clase también resulta válido.

Sin embargo, los cambios más radicales aparecen en las pruebas de integración, cuando nos fijamos en la interacción entre objetos de distintas clases. El esquema de integración basado en la existencia de una estructura jerárquica que permita unir elementos de forma ascendente o descendente no tiene sentido en las estructuras de colaboración en red definidas para el software orientado a objetos. Para realizar la integración existen propuestas que permiten encontrar aquellas clases que dependen en su funcionamiento de sí mismas o sólo de unas pocas clases para tomarlas corno destino de la primera oleada de pruebas de integración (después se irían sometiendo a prueba las clases que dependen sólo de las ya probadas en la primera tanda y así consecutivamente hasta integrar toda la aplicación).

Sin embargo, el enfoque más habitual consiste en ir probando hilos de clases que colaboran para una función o servicio del sistema (seguramente definidos en los diagramas de colaboración). Siempre habrá que asegurar todos los conjuntos de colaboración con todos los métodos de todas las clases de la aplicación. De nuevo, los diagramas de estados pueden apodar una mayor información para el rigor de la prueba.

Para terminar, señalaremos dos cuestiones que tienen una influencia capital en las pruebas de software orientado a objetos:

- Uno de los problemas que provoca la tecnología de objetos en las pruebas es el tratamiento de la herencia. Las dificultades proceden del hecho de que la prueba de un método, que es heredado en clases inferiores, suponga probar no sólo el método genérico sino también todas las implementaciones en las clases hijas y en todo el árbol de herencia que genera. Además, el polimorfismo interviene para complicar las pruebas. Hay, entonces, que generar casos de prueba adicionales para cada implementación además de los que se heredan o solapan con los de la prueba en las clases padre.

- También es conveniente recordar que la propia programación o diseño orientado a objetos (con conceptos como encapsulación, herencia, etc.) hace, en relación con los desarrollos estructurados, menos probables ciertos tipos de error (por lo que los casos de prueba dedicados a detectarlos deberán disminuir), más probables otros (que deberán ser más vigilados en las pruebas) y la aparición de nuevos tipos de defectos.