ESCUELA DE PREPARACIÓN DE OPOSITORES

E. P. O.

C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54

30001 MURCIA

INF30

Prueba y documentación de programas. Técnicas.

SAI32

Técnicas para la verificación, prueba y documentación de programas.

Esquema.

1 **INTRODUCCIÓN.** **1**

2 **EL PROCESO DE PRUEBA.** **2**

2.1 TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA 3

2.1.1 Pruebas estructurales. *4*

Utilización de la complejidad ciclomática de McCabe 5

2.1.2 Prueba funcional *7*

Particiones o clases de equivalencia. 8

Análisis de Valores Límite (AVL) 9

Conjetura de errores 10

2.1.3 Pruebas aleatorias. *10*

2.1.4 Enfoque práctico recomendado para el diseño de casos. *10*

2.2 DOCUMENTACIÓN DEL DISEÑO DE LAS PRUEBAS 11

2.2.1 Plan de pruebas. *11*

2.2.2 Especificación del diseño de pruebas. *12*

2.2.3 Especificación de caso de prueba. *12*

2.2.4 Especificación de procedimiento de prueba. *12*

2.3 EJECUCIÓN DE LAS PRUEBAS. 13

2.3.1 El proceso de ejecución. *13*

2.3.2 Documentación de la ejecución de pruebas. *13*

Histórico de pruebas. 14

Informe de incidente. 14

Informe resumen de las pruebas. 14

2.3.3 Depuración. *14*

2.4 ESTRATEGIA DE APLICACIÓN DE LAS PRUEBAS 15

3 **DOCUMENTACIÓN DE PROGRAMAS.** **15**

3.1 DOCUMENTACIÓN DEL ANÁLISIS Y EL DISEÑO 15

3.2 DOCUMENTACIÓN DEL CÓDIGO. 17

4 **CONCLUSIONES.** **19**

1 Introducción.

Una de las características típicas del desarrollo de software es la realización de controles periódicos, normalmente coincidiendo con los hitos del proyectos o la terminación de documentos. Estos controles pretenden una evaluación de la calidad de

los productos generados 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 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 comienzo 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 de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos especificados".

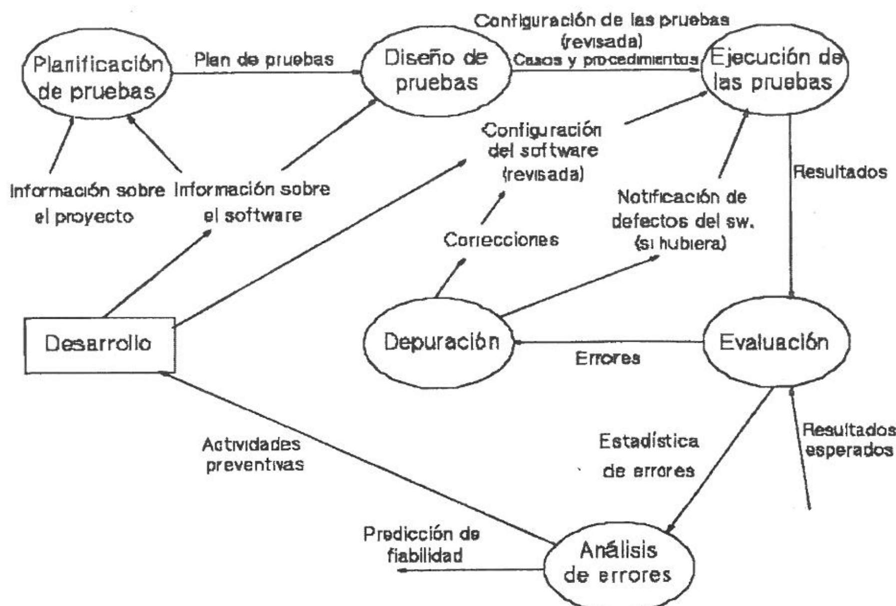
Como hemos dicho, las pruebas permiten verificar y validar el software cuando ya está en forma de código ejecutable. En este tema, vamos a estudiar las principales técnicas para la prueba del software basadas en el estándar IEEE 1008, así como para su documentación, según el estándar IEEE 829.

2 El proceso de prueba.

El objetivo de las pruebas es la detección de defectos en el software. Se trata de una actividad a posteriori, de detección, y no de prevención de problemas en el software. Las pruebas permiten la rectificación en el software. Por todo ello, el descubrimiento de un defecto significa un éxito para la mejora de la calidad.

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".

En la siguiente figura se puede ver una representación del proceso completo relacionado con las pruebas basada en parte en el estándar IEEE 1008.



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: la depuración (localización y corrección de defectos), y 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.

2.1 Técnicas de diseño de casos de prueba.

El diseño de casos de prueba está totalmente mediatizado por la imposibilidad de probar exhaustivamente el software. En consecuencia, las técnicas de diseño de casos de prueba tienen como objetivo conseguir una confianza aceptable en que se detectarán los defectos existentes sin necesidad de consumir una cantidad excesiva de recursos. Toda la disciplina de pruebas debe moverse, por lo tanto, en un equilibrio entre la disponibilidad de recursos y la confianza que aportan los casos para descubrir los defectos existentes.

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 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. Existen tres enfoques principales para el diseño de casos:

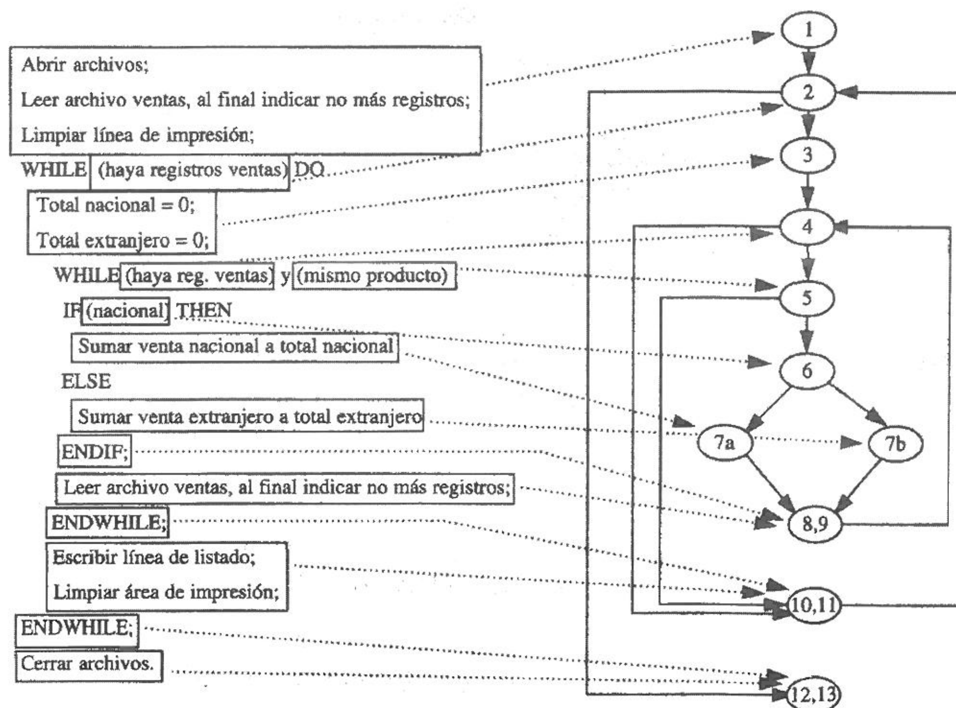
- El enfoque estructural o de caja blanca. Consiste en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba. En este caso, la prueba ideal (exhaustiva) del software consistiría en probar todos los posibles caminos de ejecución, a través de las instrucciones del código, que puedan trazarse.
- 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.
- El enfoque aleatorio consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a

partir de ellos los casos de 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.

2.1.1 Pruebas estructurales.

El diseño de casos tiene que basarse en la elección de caminos importantes que ofrezcan una seguridad aceptable en descubrir defectos, 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* o *flowcharts*). Como ejemplo de diagrama de flujo junto al código correspondiente, veamos la siguiente figura:



Una posible clasificación de criterios de cobertura lógica es la que se ofrece a continuación. 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, el 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.

- 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.

- 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.

- Cobertura de condiciones. Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez. No podemos asegurar que si se cumple la cobertura de condiciones se cumple necesariamente la de decisiones.
- Criterio de decisión/condición. Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.
- Criterio de condición múltiple. En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea (por ejemplo, según se ejecuta en el procesador), se podría considerar que cada decisión multicondicional se descompone en varias decisiones unicondicionales.

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. 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 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.

Si trabajamos con los caminos de prueba, existe la posibilidad de cuantificar el número total de caminos utilizando algoritmos basados en matrices que representan el grafo de flujo del programa. Así, es posible ofrecer diversos métodos basados en ecuaciones, expresiones regulares y matrices que permiten tanto calcular el número de caminos como enumerar dichos caminos expresados como series de arcos del grafo de flujo. 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.

Utilización de la complejidad ciclomática de McCabe.

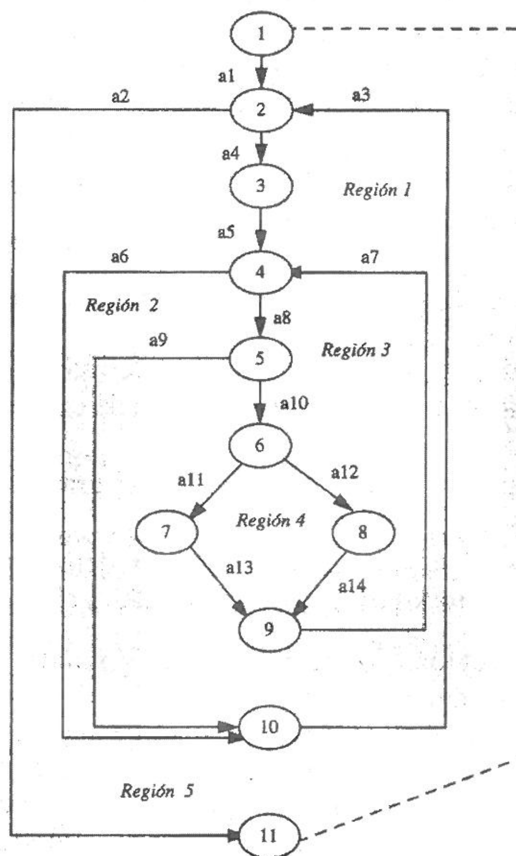
La utilización de la métrica de McCabe ha sido muy popular en el diseño de pruebas desde su creación. Esta métrica es un indicador del número de caminos independientes que existen en un grafo. El propio McCabe definió como un buen criterio de prueba la consecución de la ejecución de un conjunto de caminos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

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

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

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



1. V(G)=14-11+2=5. Los arcos han sido identificados con las marcas desde a1 hasta a14. Los nodos están numerados del 1 al 11.
2. V(G)=5. Las regiones o áreas cerradas (limitadas por aristas) del grafo son cinco. Se ha marcado un área (región 5) añadiendo un arco ficticio desde el nodo 11 al 1. Esto se debe a que las fórmulas de McCabe sólo son aplicables a aquellos grafos 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 esta condición. Por eso, debemos marcar dicho arco o, como alternativa, contabilizar la región externa al grafo como una más.
3. V(G)=4+1. Los nodos de condición son el 2, el 4, el 5 y el 6. 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) partiera 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 V(G) podemos afirmar que el número máximo de caminos independientes de dicho grafo es cinco. El criterio de prueba de McCabe consiste en elegir cinco 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.

En nuestro caso, un posible conjunto de caminos (descritos como secuencias de nodos visitados) podría ser el siguiente: 1-2-11, 1-2-3-4-10-2, 1-2-3-4-5-10-2, 1-2-3-4-5-6-7-9-4-10-2-11, 1-2-3-4-5-6-8-9-4-10-2-11. 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.

2.1.2 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, de las entradas y de las salidas. Lamentablemente, la prueba exhaustiva de caja negra también es generalmente impracticable. De nuevo, 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 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.

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, y 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 la identificación de clases de equivalencia, y la 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 partir de ellas, se identifican clases de equivalencia que pueden ser: de datos válidos o 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:
    - 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≤número≤49) y dos clases no válidas (número<1 y número>49).
    - 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≤propietarios≤3) y dos no válidas (propietarios<1 y propietarios>3).
    - 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").
    - 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).
    - 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 diseñó 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 las 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 una única clase no válida 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.

Análisis de Valores Límite (AVL).

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 los 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 las 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 se deben generar casos para los extremos del rango y casos no válidos para situaciones justo más allá de los extremos.
2. Si la condición de entrada especifica un número de valores, hay que escribir casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo de valores.
3. Usar la regla 1 para la condición de salida.
4. Usar la regla 2 para cada condición de salida. En esta regla, como en la 3, debe recordarse que:
   - Los valores límite de entrada no generan necesariamente los valores límite de salida.
   - No siempre se pueden generar resultados fuera del rango de salida (pero es interesante considerarlo).
5. Si la entrada o la salida de un programa es un conjunto ordenado, los casos se deben concentrar en el primero y en el último elemento.

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:

- El valor cero es una situación propensa a error tanto en la salida como en la entrada.
- En situaciones en las que se introduce un número variable de valores, conviene centrarse en el caso de no introducir ningún valor y en el de un solo valor. También puede ser interesante introducir todos los valores iguales.
- Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación.
- También interesa imaginar lo que el usuario puede introducir como entrada a un programa. Se dice que se debe preveer toda clase de acciones de un usuario como si fuera "completamente tonto" o, incluso, como si quisiera sabotear el programa.

2.1.3 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 sin parar. Esto implica usar una herramienta denominada un 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.

Si el proceso de generación se ha realizado correctamente, se crearán eventualmente 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 técnicas de caja blanca y de caja negra.

2.1.4 Enfoque práctico recomendado para el diseño de casos.

Los enfoques estudiados 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.

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.
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.
6. Examinar la lógica del programa para añadir los casos precisos (de caja blanca) para cumplir el criterio de cobertura elegido sí los resultados de la ejecución del punto 5 indican que no se ha satisfecho el criterio de cobertura elegido.

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.

Debemos recordar que tanto la prueba exhaustiva de caja blanca como de caja negra son impracticables. Conviene emplear lo mejor de todas las técnicas para obtener pruebas más eficaces.

2.2 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, como vimos, 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 829, 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.
- Tras generar los casos de prueba detallados, se debe especificar cómo proceder en detalle en la ejecución de dichos casos.
- Tanto las especificaciones de casos de prueba como 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.

2.2.1 Plan de pruebas.

El objetivo del documento es 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. Para ello, el estándar fija la siguiente estructura:

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.

2.2.2 Especificación del diseño de pruebas.

El objetivo del documento es 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. Para ello, el estándar fija la siguiente estructura:

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, y 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).

2.2.3 Especificación de caso de prueba.

El objetivo del documento es definir uno de los casos de prueba identificado por una especificación del diseño de las pruebas. Para ello, el estándar fija la siguiente estructura:

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).
4. Especificaciones de todas las salidas y las características requeridas para los elementos que se van a probar.
5. Necesidades de entorno.
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).

2.2.4 Especificación de procedimiento de prueba.

El objetivo del documento es especificar los pasos para la ejecución de un conjunto de casos de prueba o, más generalmente, los pasos utilizados para analizar un elemento software con el propósito de evaluar un conjunto de características del mismo. Para ello, el estándar fija la siguiente estructura:

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.
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, las acciones necesarias para empezar la ejecución, las acciones necesarias durante la ejecución, cómo se realizarán las medidas, las acciones necesarias para suspender la prueba, los puntos para reinicio de la ejecución y acciones necesarias para el reinicio en estos puntos, las acciones necesarias para detener ordenadamente la ejecución, las acciones necesarias para restaurar el entorno y dejarlo en la situación existente antes de las pruebas, y las acciones necesarias para tratar los acontecimientos anómalos.

2.3 Ejecución de las pruebas.

2.3.1 El proceso de ejecución.

El proceso de las pruebas, según el estándar IEEE 1008, en el que se incluye la ejecución de pruebas abarca las siguientes fases:

1. Ejecutar las pruebas, cuyos casos y procedimientos han sido ya diseñados previamente.
2. Comprobar si se ha concluido el proceso de prueba (según ciertos criterios de completitud de prueba que suelen especificarse en el plan de pruebas).
3. 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 completitud de pruebas.

Los criterios de completitud 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 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 las posibles excepciones aprobadas en el plan de pruebas).

2.3.2 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.

En el estándar IEEE 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.
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, e informes de los incidentes ocurridos (si hay)

durante la ejecución. La documentación de salida correspondiente a un mismo diseño de prueba se recoge en un informe resumen de pruebas.

A continuación, veremos el formato de los distintos documentos descritos por el estándar IEEE 829.

Histórico de pruebas.

El histórico de pruebas (*test log*) documenta todos los hechos relevantes ocurridos durante la ejecución de las pruebas. La estructura fijada en el estándar es:

1. Identificador.
2. Descripción de la prueba: elementos probados y entorno de la prueba.
3. Anotación de datos sobre cada hecho ocurrido (incluido el comienzo y el final de la prueba): fecha y hora, e identificador de informe de incidente.
4. Otras informaciones.

Informe de incidente.

El informe de incidente (*test incident report*) documenta cada incidente ocurrido en la prueba y que requiera una posterior investigación. La estructura fijada en el estándar es:

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

Informe resumen de las pruebas.

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. La estructura fijada en el estándar es:

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.

2.3.3 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: encontrar la causa del error, analizarla y corregirla; y 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 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.

2.4 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 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: comienzan a nivel de módulo; una vez terminadas, progresan hacia la integración del sistema completo y su instalación; y 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).

3 Documentación de programas.

3.1 Documentación del análisis y el diseño.

Se puede usar el esquema de documento de la siguiente tabla como modelo para la especificación de diseño. Cada Sección está compuesta por varios párrafos que se centran en los diferentes aspectos.

- I. Ámbito
  - A. Objetivos del sistema
  - B. Hardware, software e interfaces humanas
  - C. Principales funciones del software
  - D. Base de datos definida externamente
  - E. Principales restricciones y limitaciones del diseño
- II. Documentos de referencia
  - A. Documentación de software existente
  - B. Documentación del sistema
  - C. Documentos del vendedor (hardware o software)
  - D. Referencia técnica
- III. Descripción del diseño
  - A. Descripción de datos
    - 1. Revisión del flujo de datos
    - 2. Revisión de la estructura de datos
  - B. Estructura de programa derivada
  - C. Interfaces dentro de la estructura
- IV. Módulos: *para cada módulo:*
  - A. Texto explicativo
  - B. Descripción de la interfaz
  - C. Descripción en lenguaje de diseño
  - D. Módulos usados
  - E. Organización de los datos
  - F. Comentarios
- V. Estructura de archivos y datos globales
  - A. Estructura de archivos externos
    - 1. Estructura lógica
    - 2. Descripción lógica de los registros
    - 3. Método de acceso
  - B. Datos globales
  - C. Referencias cruzadas entre archivos y datos
- VI. Referencias cruzadas para los requisitos (véase la figura 10.25)
- VII. Provisiones de prueba
  - A. Directrices de prueba
  - B. Estrategia de integración
  - C. Consideraciones especiales
- VIII. Empaquetamiento
  - A. Provisiones especiales de solapamiento del programa
  - B. Consideraciones de transferencia
- IX. Notas especiales
- X. Apéndices

Las secciones de la especificación de diseño se irán completando conforme se refine la representación del software.

El ámbito global del trabajo se describe en la Sección I. Las referencias específicas a la documentación de soporte se incluyen en la Sección II.

La Sección III, la descripción del diseño, se desarrolla durante el diseño preliminar. El diseño está conducido por la información, esto es, la estructura y/o flujo de los datos dictan la arquitectura del software. En esta sección, para formar la estructura del software, se refinan y utilizan los diagramas de flujo de datos y otras representaciones de los datos, desarrollados durante el análisis de requisitos. Puesto que está disponible el flujo de información, se pueden desarrollar las descripciones de la interfaz para los elementos del software.

Las secciones IV y V se realizan a medida que nos movemos del diseño preliminar al diseño detallado. Inicialmente, se describen los módulos (elementos del software referenciables por separado, tales como subrutinas, funciones y procedimientos) mediante texto descriptivo en lenguaje natural. El texto explica la

función procedimental del módulo. Después, se usa una herramienta de diseño procedimental para traducir el texto explicativo a una descripción estructurada.

La Sección V contiene una descripción de la organización de los datos. Durante el diseño preliminar se describen las estructuras de archivo, mantenidas en memoria secundaria; se asignan los datos globales y se establecen referencias cruzadas que conectan los módulos individuales con los archivos o datos globales.

La Sección VI de la especificación de diseño contiene referencias cruzadas para los requisitos. El propósito de la matriz de referencias cruzadas es: (1) asegurar que todos los requisitos se satisfacen en el diseño del software; (2) indicar qué módulos son críticos para la implementación de requisitos específicos.

En la Sección VII del documento de diseño se encuentra la primera etapa del desarrollo del procedimiento de prueba. Una vez que se ha establecido la estructura y las interfaces del software, podemos desarrollar directrices para la prueba de los módulos individuales y de su integración en paquetes. En algunos casos, la especificación detallada de los procedimientos de prueba se lleva a cabo en paralelo con el diseño. En tales casos, se puede suprimir esta sección de la especificación de diseño.

Las restricciones del diseño, tales como las limitaciones físicas de memoria o la necesidad de un gran rendimiento, pueden dictaminar requisitos especiales para el ensamblado o empaquetamiento del software. Algunas consideraciones especiales producidas por la necesidad de solapar programas, por la necesidad de gestión virtual de memoria o de procesamiento a alta velocidad u otros factores, pueden introducir modificaciones en el diseño obtenido del flujo o de la estructura de la información. Los requisitos y consideraciones para el empaquetamiento del software se incluyen en la Sección VIII. Secundariamente, esta sección describe el método que se usará para transferir el software al lugar del cliente.

Las secciones IX y X de la especificación contienen datos complementarios. Las descripciones de algoritmos o procedimientos alternativos, los datos tabulares, los extractos de otros documentos y otras informaciones relevantes se incluyen como notas especiales o como apéndices aparte. Puede ser aconsejable desarrollar un manual de operación/instalación preliminar e incluirlo como apéndice del documento.

3.2 Documentación del código.

La documentación interna del código fuente comienza con la elección de los nombres de los identificadores (variables y etiquetas), continúa con la localización y la composición de los comentarios y termina con la organización visual del programa.

La elección de nombres de identificadores significativos es crucial para la legibilidad. Los lenguajes que limitan la longitud de los nombres de las variables o de las etiquetas a unos pocos caracteres, implícitamente limitan la comprensión.

Se puede argumentar que las expresiones llenas de palabras oscurecen el flujo lógico y hacen más difíciles las modificaciones. Obviamente hay que aplicar el sentido común al seleccionar los identificadores. Los identificadores innecesariamente largos pueden ser una fuente potencial de error. Sin embargo, ciertos estudios indican que, incluso para pequeños programas, la elección de identificadores significativos mejora la comprensión.

La posibilidad de expresar los comentarios en lenguaje natural como parte del listado del código fuente es algo que aparece en todos los lenguajes de programación de

propósito general. Sin embargo, surgen ciertas cuestiones: ¿cuántos comentarios son suficientes?, ¿dónde se deben situar?, ¿oscurecen la lógica del programa?, ¿pueden distraer al lector?, ¿son "no mantenibles", y, por tanto, no fiables?.

Hay pocas respuestas definitivas a las preguntas anteriores. Pero una está clara: el software debe contener documentación interna. Los comentarios permiten al programador comunicarse con otros lectores del código fuente. Los comentarios pueden resultar una clara guía de comprensión durante la última fase de la ingeniería del software: el mantenimiento.

Existen muchos modelos propuestos para la creación de comentarios. Los comentarios de prólogo y los comentarios descriptivos son dos categorías que requieren enfoques algo diferentes. Al principio de cada módulo debe haber un comentario de prólogo. El formato para esos comentarios es:

1. Una sentencia de propósito que indique la función del módulo.
2. Una descripción de la interfaz que incluya: un ejemplo de "secuencia de llamada", una descripción de todos los argumentos, y una lista de todos los módulos subordinados.
3. Una explicación de los datos pertinentes, tales como las variables importantes y su uso, de las restricciones y limitaciones y de otra información importante.
4. Una historia del desarrollo que incluya: el diseñador del módulo (autor), el revisor (auditor) y la fecha, y fechas de modificación y de descripción.

Los comentarios descriptivos se incluyen en el cuerpo del código fuente y se usan para describir las funciones de procesamiento. Los comentarios deben proporcionar algún extra, no sólo parafrasear el código. Además, los comentarios descriptivos deben:

- Describir los bloques de código en lugar de comentar cada línea.
- Usar líneas en blanco o tabulaciones de forma que sean fácilmente distinguibles del código.
- Que sean correctos; un comentario incorrecto o que se pueda interpretar mal es peor que no ponerlo.

Con unos recursos mnemotécnicos apropiados para los identificadores y unos buenos comentarios, se asegura una documentación interna adecuada.

Cuando se representa un diseño de procedimientos detallado mediante un lenguaje de diseño de programas, se puede incluir directamente la documentación del diseño en el listado fuente como sentencias de comentario. Esta técnica ayuda a asegurar que, tanto el código como el diseño, podrán fácilmente mantenerse al hacer cambios sobre ellos.

La forma en que el código fuente aparece en el listado es una importante contribución a la legibilidad. El sangrado del código fuente realza las construcciones lógicas y los bloques de código, tabulando desde el margen izquierdo de forma que se vean desplazados esos atributos. Al igual que los comentarios, no está clara cuál es la mejor forma de sangrar. El sangrado manual puede llegar a ser complicado al tener que modificar el código y algunos experimentos indican que sólo da una mejora marginal en la inteligibilidad. Probablemente, la mejor aproximación sea usar una herramienta de formato automático de código que sangre adecuadamente el código fuente. Al eliminar

el exceso de sangrado del codificador, el formulario puede ser mejorado con relativamente poco esfuerzo.

4 Conclusiones.

El objetivo de las pruebas es la detección de defectos en el software. Se trata de una actividad a posteriori, de detección, y no de prevención de problemas en el software. Las pruebas permiten la rectificación en el software.

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.

El diseño de casos de prueba está totalmente mediatizado por la imposibilidad de probar exhaustivamente el 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. La dificultad de esta idea es saber elegir los casos que se deben ejecutar. Existen tres enfoques principales para el diseño de casos: El enfoque estructural o de caja blanca, el enfoque funcional o de caja negra, y el enfoque aleatorio.

La documentación de las pruebas es necesaria para una buena organización de las mismas, así como para asegurar su reutilización. Hemos estudiado los distintos documentos de trabajo de las pruebas, según el estándar IEEE 829. Éstos son: el plan de pruebas, la especificación del diseño de pruebas, la especificación de caso de prueba, y la especificación de procedimiento de prueba.

El proceso de las pruebas, según el estándar IEEE 1008, en el que se incluye la ejecución de pruebas abarca las siguientes fases: ejecutar las pruebas, comprobar si se ha concluido el proceso de prueba, y evaluar los resultados o generar las pruebas adicionales para que se satisfagan los criterios de completitud de 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. En el estándar IEEE 829 se pueden distinguir dos grupos principales: documentación de entrada, y documentación de salida o informes sobre la ejecución. Los documentos descritos por el estándar IEEE 829 son: el histórico de pruebas, el informe de incidente, y el informe resumen de las pruebas.

La depuración es el proceso de localizar, analizar y corregir los defectos que se sospecha que contiene el software. Las dos principales etapas en la depuración son: la localización del defecto, y la corrección del defecto.

La estrategia de pruebas suele seguir las siguientes etapas: comienzan a nivel de módulo; una vez terminadas, progresan hacia la integración del sistema completo y su instalación; y culminan cuando el cliente acepta el producto y se pasa a su explotación inmediata.

La documentación interna del código fuente comienza con la elección de los nombres de los identificadores (variables y etiquetas), continúa con la localización y la composición de los comentarios y termina con la organización visual del programa.