Sistemas y Aplicaciones Informáticas

Tema 32. Técnicas para la Verificación, Prueba y Documentación de Programas.

1. ÁMBITO DE DOCENCIA. 3

2. TÉCNICAS PARA LA VERIFICACIÓN Y PRUEBA DE PROGRAMAS. 3

2.1. INTRODUCCIÓN. 3
2.1.1. Ciclo de vida del software. Conceptos de error, defecto, fallo y prueba. 3
2.1.2. Fases del ciclo de pruebas. Verificación y validación. 3
2.2. DISEÑO DE LAS PRUEBAS. 3
2.2.1. Diseño de casos de prueba. Recomendaciones y enfoques. 3
2.2.2. Enfoque estructural o de caja blanca. 4
2.2.3. Enfoque funcional o de caja negra. 5
2.2.3.1. Particiones de equivalencia. 5
2.2.3.2. Análisis de valores límite. 6
2.2.3.3. Conjetura de errores. 6
2.3. EJECUCIÓN DE LAS PRUEBAS (IEEE 1008). 6
2.3.1. Proceso y estrategia de ejecución de pruebas. Depuración. 6
2.3.2. Pruebas unitarias. 7
2.3.3. Pruebas de integración. 7
2.3.4. Pruebas de validación. 8
2.3.5. Pruebas de sistema. 8

3. DOCUMENTACIÓN DE PROGRAMAS. 8

3.1. DOCUMENTACIÓN FORMAL DE UN PROYECTO. 8
3.2. DOCUMENTACIÓN DE ESPECIFICACIONES (IEEE 830). 9
3.3. DOCUMENTACIÓN DE DESARROLLO. 9
3.4. DOCUMENTACIÓN DE PRUEBAS (IEEE 829). 10

1. Ámbito de docencia.

- Sistemas informáticos monousuario y multiusuario (ASI 1).
- Sistemas informáticos multiusuario y en red (DAI 1).
- Sistemas operativos en entornos monousuario y multiusuario (ESI 1).

2. Técnicas para la verificación y prueba de programas.

2.1. Introducción.

2.1.1. Ciclo de vida del software. Conceptos de error, defecto, fallo y prueba.

- El ciclo de vida del software consta de análisis, diseño, desarrollo, pruebas, implantación y mantenimiento. La fase de pruebas es crítica para la garantía de la calidad del software y representa la revisión final de las especificaciones, del diseño y del desarrollo.
- Conceptos asociados:
  * *Error*. Es una acción humana que conduce a un resultado incorrecto.
  * *Defecto*. Es un paso de procesamiento o una definición de datos incorrectos en un programa.
  * *Fallo*. Es la incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados.
  * *Prueba*. Es la ejecución controlada de un programa con el fin de encontrar defectos.
- Los errores cometidos en las fases anteriores se plasman en defectos del software, que dan lugar a fallos del sistema. Los efectos negativos de los fallos dependen de la criticidad del sistema.
- Por tanto, el software debe comprobarse mediante pruebas con el fin de conseguir que el programa funcione correctamente y se descubran los defectos. Se pretende que con el mínimo esfuerzo posible se detecten el mayor número posible de defectos.

2.1.2. Fases del ciclo de pruebas. Verificación y validación.

- El ciclo completo de las pruebas pasa por las siguientes fases:
  * *Planificación de las pruebas*. Su objetivo es elaborar un plan de pruebas basándose en la información sobre el proyecto y sobre el software.
  * *Diseño de las pruebas*. En esta fase se detalla la configuración de las pruebas, con los casos y los procedimientos de prueba.
  * *Ejecución de las pruebas*. La aplicación de los casos de prueba da lugar a unos resultados.
  * *Evaluación y depuración*. Consiste en comparar los resultados producidos con los esperados, y localizar y corregir los defectos del software para eliminar fallos.
- **Verificación**. Es el conjunto de actividades que aseguran que el software implementa correctamente una función específica. Responde a la pregunta ¿estamos construyendo correctamente el producto?
- **Validación**. Es el conjunto de actividades que aseguran que el software se ajusta a los requisitos del cliente. Responde a la pregunta ¿estamos construyendo el producto correcto?

2.2. Diseño de las pruebas.

2.2.1. Diseño de casos de prueba. Recomendaciones y enfoques.

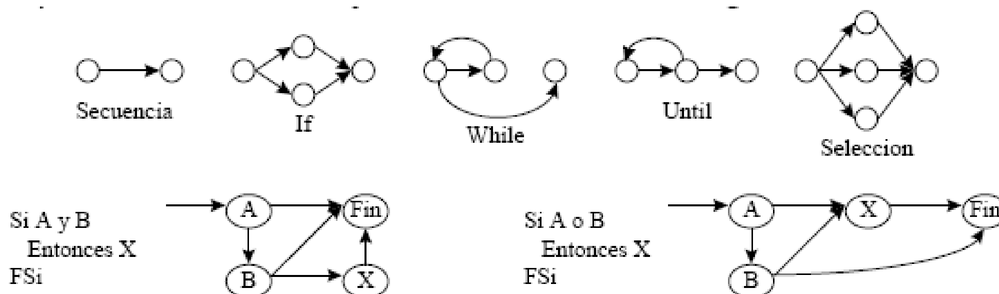
- La prueba exhaustiva del software es impracticable, no se pueden probar todas las posibilidades de su funcionamiento ni siquiera en programas sencillos. Por tanto, debe realizarse la

planificación y el diseño de los casos de prueba, que tiene como objetivo conseguir una confianza aceptable sin necesidad de consumir una gran cantidad de recursos.

– Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular. Cada caso de prueba debe definir el resultado de salida esperado, que se comparará con el realmente obtenido.
– Recomendaciones en el diseño de los casos de prueba:
  * El programador debe evitar probar sus propios programas, ya que es normal que las situaciones que olvidó al crear el programa se olviden también al crear los casos de prueba.
  * Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.
  * Las pruebas deben centrarse en probar si el sistema no hace lo que debe hacer, y si hace lo que no debe hacer, es decir, si provoca efectos secundarios adversos.
  * Se deben evitar los casos no documentados ni diseñados con cuidado, ya que es necesario probar muchas veces el software, y se debe tener claro lo que funciona y lo que no funciona.
  * No deben hacerse planes de prueba suponiendo que prácticamente no hay defectos en los programas y dedicando pocos recursos a las pruebas, puesto que siempre hay defectos.
  * Donde hay un defecto hay otros, es decir, la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubierto.

2.2.2. Enfoque estructural o de caja blanca.

– Consiste en centrarse en la estructura interna del programa para elegir los casos de prueba, analizando los caminos de ejecución. El diseño de los casos de prueba debe basarse en la elección de diferentes flujos de ejecución del programa con el fin de detectar el máximo número de errores. Para ello se utiliza el criterio de cobertura lógica, que utiliza la representación de un grafo de flujo para establecer los caminos posibles desde la sentencia inicial hasta la final.
– El diagrama de flujo consiste en un grafo dirigido en el que los nodos están formados por bloques de sentencias simples que incluyen alguna condición, y los arcos representan el flujo de ejecución del programa. Si existe una combinación de condiciones, se separan en el propio grafo.



– Los criterios de cobertura lógica se clasifican en orden ascendente de exigencia y coste así:
  * *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 diseñar casos de prueba para que al menos cada decisión lógica tenga un resultado verdadero y otro falso.

* *Cobertura de condiciones*. Consiste en generar los casos de prueba para que cada condición de cada decisión tome al menos una vez un valor verdadero y otro falso.
* *Criterio de condición múltiple*. Consiste en diseñar casos de prueba para que se cumplan de forma independiente las condiciones de una condición múltiple.
* *Criterio de cobertura de caminos*. Se recorren todos los caminos (impracticable).

- Un camino independiente es cualquier camino del programa que introduce por lo menos un nuevo conjunto de sentencias o una nueva condición. La complejidad ciclomática $V(G)$ es un indicador del número de caminos independientes que existen en un grafo. Se calcula así:

  $V(G) = a - n + 2$, siendo a el número de arcos del grafo y n el número de nodos
- El valor de $V(G)$ proporciona el límite superior para el número de caminos independientes que componen el programa, y por tanto también da el límite superior para el número de pruebas que deben diseñarse para garantizar que se cubren todas las sentencias del programa. Cuando $V(G)$ es mayor que 10 la probabilidad de defectos en el módulo o programa crece mucho. En este caso quizás sea interesante dividir el programa en varios módulos.

2.2.3. Enfoque funcional o de caja negra.

2.2.3.1. Particiones de equivalencia.

- Consiste en estudiar la especificación de las funciones del programa, la entrada y la salida para crear los casos. Los casos de prueba deben ejecutar el máximo número de posibilidades de entrada diferentes para así reducir el total de casos. Para ello debe dividirse el dominio de valores de entrada en un número finito de clases de equivalencia, de manera que 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.
- Una clase de equivalencia es un conjunto de estados válidos o inválidos para los valores de entrada, que pueden ser un valor específico, un rango de valores, un conjunto de valores relacionados o una condición lógica.
- En primer lugar deben identificarse las clases de equivalencia y crear los casos de prueba correspondientes. Los pasos para identificar clases de equivalencia son los siguientes:
  * Identificación de las condiciones de las entradas del programa, es decir, restricciones de formato o contenido de los datos de entrada.
  * A partir de ellas, se identifican clases de equivalencia que pueden ser de datos válidos y de datos no válidos o erróneos.
  * Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:
    - Si se especifica un rango de valores para los datos de entrada, deberá crearse una clase válida y dos clases inválidas.
    - Si se especifica un valor específico, deberá crearse una clase válida y dos inválidas.
    - Si se especifica un miembro de un conjunto de valores relacionados, deberán identificarse por cada valor una clase válida y una no válida.
    - Si se especifica una condición lógica, se identificarán una clase válida y una no válida.
    - En cualquier caso, si ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

* A continuación, deben utilizarse las clases de equivalencia para identificar los casos de prueba correspondientes. Este proceso consta de las siguientes fases:
    - Asignación de un número único a cada clase de equivalencia.
    - Habría que diseñar casos de prueba que cubran todas las clases de equivalencia, tanto válidas como inválidas. En el caso de las clases de equivalencia inválidas, deben diseñarse casos de prueba distintos.
    - Hasta que todas las clases de equivalencia válidas hayan sido cubiertas por los casos de prueba, escribir un caso que cubra tantas clases válidas no cubiertas como sea posible.
    - Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por los casos de prueba, escribir un caso para una única clase no válida sin cubrir.

2.2.3.2. Análisis de valores límite.

- El análisis de valores límite es una técnica de diseño de casos de prueba que complementa a la de particiones de equivalencia. Las diferencias son las siguientes:
    * Además de elegir cualquier elemento como representativo de una clase de equivalencia, se selecciona uno o más elementos para que los márgenes se sometan a prueba.
    * Además de concentrarse únicamente en el dominio de entrada, los casos de prueba se generan considerando también el espacio de salida.
- Las reglas para identificar los valores límite son las siguientes:
    * Si una condición de entrada especifica un rango de valores, se deben generar casos válidos para los extremos del rango, y casos no válidos para situaciones justo por debajo y por encima de los extremos inferior y superior respectivamente.
    * Si la condición de entrada especifica un número finito y consecutivo de valores, hay que escribir casos válidos para los números mínimo y máximo, y casos no válidos para situaciones justo por debajo y por encima del mínimo y del máximo respectivamente.
    * Aplicar estos mismos criterios, pero para los valores de salida, aunque no siempre se pueden generar resultados fuera del rango de salida.
    * 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.

2.2.3.3. Conjetura de errores.

- 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 una lista que tiene todos los valores iguales.
- Es recomendable imaginar que el programador pudiera haber interpretado algo mal en la especificación, y también lo que el usuario puede introducir como entrada a un programa.
- El valor cero es una situación propensa a error tanto en la salida como en la entrada.

2.3. Ejecución de las pruebas (IEEE 1008).

2.3.1. Proceso y estrategia de ejecución de pruebas. Depuración.

- El proceso de ejecución de pruebas consta de los siguientes pasos:
    * Ejecutar los casos de prueba diseñados. Si existe algún fallo provocado por defectos en las pruebas o en el software, eliminar dichos defectos (depuración) y volver a probar.

* Comprobar si han concluido los casos de prueba. Si no es así realizar las pruebas adicionales y ejecutarlas. En caso contrario, evaluar los resultados y realizar las depuraciones oportunas.

– La estrategia de ejecución de pruebas integra el diseño de los casos de prueba en una serie de casos coordinados a través de la creación de distintos niveles de prueba con distintos objetivos cado uno. Existe una correspondencia entre cada nivel de prueba y cada etapa del desarrollo:
  * *Especificación y lógica de módulos → Pruebas unitarias*. Centran 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).
  * *Diseño modular → Pruebas de integración*. Deben tener en cuenta los mecanismos de agrupación de módulos, fijados en la estructura del programa, y en general los interfaces entre los componentes de la arquitectura del software.
  * *Requisitos de usuario → Pruebas de validación*. Sirven para que el usuario pueda verificar si el producto se ajusta a los requisitos establecidos por él mismo.
  * *Especificación de requisitos → Pruebas de sistema*. Comprueban los desajustes entre el software y los requisitos de funcionamiento, y el grado de cumplimiento de los objetivos.

– La depuración es el resultado de una prueba efectiva. Su objetivo es encontrar y corregir la causa del error. Como estrategia se puede utilizar:
  * *Fuerza bruta*. El error se encuentra mediante la traza de la ejecución y sentencias WRITE.
  * *Vuelta atrás*. Se recorre manualmente el código hacia atrás hasta encontrar el error.
  * *Eliminación de causas*. Se establecen los datos relacionados con el error y se agrupan para determinar una hipótesis de causa. Se establecen pruebas para desechar grupos de datos y se van probando hasta aislar los datos que producen el error.

2.3.2. Pruebas unitarias.

– Se trata de las pruebas formales que permiten asegurar que cada módulo funciona individualmente de forma adecuada, y son de tipo caja blanca. Puede abarcar desde un módulo hasta un grupo de módulos, incluso un programa completo. Estas pruebas suelen realizarlas el propio personal de desarrollo, pero evitando que sea el propio programador del módulo.

– Los aspectos a evaluar son los siguientes:
  * *Interfaz*. Comprobar la información que entra y sale del módulo.
  * *Estructuras de datos locales*. Comprobar la integridad de los datos locales.
  * *Caminos independientes*. Asegurar que todos los caminos y cálculos funcionan bien.
  * *Caminos de manejo de errores*. Disponer de unos caminos de manejo de errores que informen y gestionen el procesamiento cuando se detecta un error.
  * *Condiciones límite*. Comprobar que el módulo funciona correctamente en los límites establecidos como restricciones de procesamiento.

2.3.3. Pruebas de integración.

– Su objetivo es verificar el diseño y la construcción del programa, y son del tipo caja negra más algunas de caja blanca. Implican una progresión ordenada de pruebas que van desde los componentes o módulos culminando en el sistema completo. Normalmente se solapan con las pruebas unitarias y las realiza el desarrollador.

- Hay dos tipos fundamentales de integración:
  - *Integración no incremental*. Se prueba cada módulo por separado y luego se integran todos de una vez y se prueba el programa completo.
  - *Integración incremental*. Se combina el siguiente módulo que se debe probar con el conjunto de módulos que ya han sido probados. Hay tres maneras de realizar la integración de la jerarquía arborescente de módulos:
    - **Ascendente**. Comenzando por los módulos hoja, se utilizan módulos denominados conductores que simulan la llamada para introducir los datos de prueba y recoger los resultados. Una vez probados, los módulos conductores se van sustituyendo sucesivamente por los módulos del nivel superior de jerarquía. El problema es que el programa como entidad no existe hasta el final.
    - **Descendente**. Comienza por el módulo de control principal y va incorporando módulos subordinados, descendiendo progresivamente en profundidad o en amplitud. Se utilizan módulos denominados resguardo que simulan los módulos dependientes que aún faltan por probar, los cuales implican más esfuerzo. Si hay secciones críticas especialmente complejas deben integrarse lo antes posible. El orden de integración debe incorporar cuanto antes los módulos de entrada/salida para facilitar la ejecución de pruebas.
    - **Sandwich**. Es una combinación de la estrategia descendente para los módulos de jerarquía superior, y de la estrategia ascendente para los módulos de jerarquía inferior.

2.3.4. Pruebas de validación.

- Son pruebas en las que participa el usuario, y están enfocadas hacia la comprobación de los requisitos especificados. En el caso del software a medida, son pruebas de tipo caja negra planificadas y organizadas formalmente para determinar si se cumplen los requisitos de aceptación marcados por el cliente.
- En el caso de productos comerciales son las pruebas alfa, realizadas por un cliente en un entorno controlado con el desarrollador delante, y las pruebas beta, que se llevan a cabo en varios lugares por usuarios finales en un entorno normal sin apoyo ninguno.

2.3.5. Pruebas de sistema.

- Es el proceso de prueba de un sistema integrado de hardware y software. Son de tipo caja negra y permiten comprobar el cumplimiento de todos los requisitos funcionales, considerando el producto software final al completo en un entorno de sistema.
- Permiten verificar además la seguridad del sistema y su capacidad de recuperación de fallos, y el funcionamiento y rendimiento en las interfaces hardware, software, de usuario y de operador en condiciones límite y de sobrecarga.

3. Documentación de programas.

3.1. Documentación formal de un proyecto.

- **Oferta de desarrollo**. Recoge una descripción del acuerdo alcanzado por la empresa proveedora y la empresa cliente. Deben especificarse valoraciones económicas, valoraciones temporales, disposiciones técnicas y una referencia al documento de especificaciones de requisitos.

- **Documentación de las especificaciones**. Recoge qué es lo que hay que hacer, con qué se debe hacer, cuándo debe estar finalizado y cómo hacerlo.
- **Documentación del desarrollo**. Recoge toda la información necesaria que se va generando durante el desarrollo de la aplicación. No se entrega al cliente.
- **Documentación de las pruebas**. Recoge todos los casos de prueba que se han generado para toda la aplicación. Se entrega al cliente cuando finaliza el desarrollo de la aplicación y deberá contener los estados del control de cambios (solucionadas, en observación, rechazadas y motivo) y de las mejoras impuestas por el cliente fuera del documento de especificaciones.
- **Documentación del cliente**. Deberá entregarse:
  - *Manual de explotación*. Recoge la información necesaria para poner en explotación la aplicación. Va dirigido a la instalación y configuración fundamentalmente, por lo que su estructura depende de la organización donde se vaya a implantar la aplicación.
  - *Manual de administración*. Es el documento que recoge las tareas que deben realizarse para el mantenimiento y administración de aplicación.
  - *Manual de usuario*. Reúne la información, normas y documentación necesaria para que el usuario conozca y utilice adecuadamente la aplicación desarrollada. El usuario debe comprender la información, ya que debe utilizarlo como guía de consulta y referencia. Los objetivos que se persiguen son el correcto uso de la aplicación y que permita detectar y corregir errores. Su redacción debe ser concisa y clara, con los apoyos gráficos necesarios.
  - *Auditoria*. Documento final que se genera para comprobar la calidad de la aplicación además de ver si cumple con la metodología de desarrollo y especificaciones de requisitos.

3.2. Documentación de especificaciones (IEEE 830).

- Este documento tiene como objeto asegurar que tanto el desarrollador como el cliente tienen la misma idea sobre las funcionalidades del sistema. Es muy importante que esto quede claro ya que de lo contrario el desarrollo software no será aceptable.
- Debe contener los siguientes apartados:
  - Fines y objetivos del software.
  - Descripción detallada del flujo y contenido de la información y de la interfaz del sistema.
  - Descripción funcional incluidas las restricciones de diseño y los requisitos de rendimiento.
  - Descripción del comportamiento del software ante sucesos externos y controles internos.
  - Criterios de validación, con clases de pruebas y respuesta esperada del software.

3.3. Documentación de desarrollo.

- Por norma general se debe documentar el código fuente, cómo se producirán las entradas y salidas, y la declaración y el uso de los datos.
- **Especificaciones de operaciones de entrada/salida**. Hay que establecer cuáles son los dispositivos de E/S que se van a utilizar, y el formato de los datos de entrada y de salida. Si el usuario tiene un alto grado de interacción con la aplicación, el interfaz ofrecido al usuario será simple e intuitivo. En cualquier operación de E/S es importante que se pueda dar marcha atrás y si se produce un error, saber cuál es el módulo que lo produjo y su posible solución.

- **Documentación del código fuente**. El nombre del programa o módulo y los nombres de las variables y estructuras de datos deben guardar relación con el contenido que tendrán para mejorar la legibilidad, manteniendo un diccionario de datos. También es importante introducir comentarios que expresen las funciones desarrolladas en el código. Deben incluirse líneas en blanco, tabuladores y sangrías que faciliten la lectura. Todo programa debe llevar la fecha de creación y fecha de las modificaciones, así como los cambios que se produzcan y su autor.

3.4. Documentación de pruebas (IEEE 829).

- Es necesaria para una buena organización de las mismas, así como para asegurar su reutilización. También contribuye a la eficacia y eficiencia de la aplicación generada.
- Los siguientes documentos están asociados a la <u>fase de diseño</u> de las pruebas:
  - *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, los criterios, las actividades de prueba, el personal responsable y los riesgos asociados.
  - *Especificación del diseño de pruebas*. El objetivo del documento es especificar los refinamientos necesarios sobre el enfoque general reflejado en el plan de pruebas e identificar las características de los elementos software que se deben probar y los criterios de paso/fallo de la prueba.
  - *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, detallando los elementos software a probar, las entradas y salidas del caso de prueba y los requisitos especiales del procedimiento de prueba.
  - *Especificación de procedimiento de prueba*. El objetivo del documento es especificar las acciones necesarias 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.
- Los siguientes documentos están asociados a la <u>fase de ejecución</u> de las pruebas:
  - *Histórico de pruebas*. El objetivo del documento es documentar todos los hechos relevantes ocurridos durante la ejecución de las pruebas.
  - *Informe de incidente*. El objetivo del documento es documentar 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.
  - *Informe resumen de pruebas*. El objetivo del documento es resumir los resultados de las actividades de prueba (las señaladas en el propio informe) y aportar una evaluación del software basada en dichos resultados.