

TEMA 7:

PRUEBAS DEL SOFTWARE Y CONTROL DE CALIDAD

ÍNDICE

1.- INTRODUCCIÓN	1
1.1.- OBJETIVOS DE LA PRUEBA	1
1.2.- ATRIBUTOS DE UNA BUENA PRUEBA	1
2.- TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA	1
2.1.- PRUEBA DE CAJA NEGRA	2
2.1.1.- PARTICIÓN EQUIVALENTE.....	2
2.1.2.- ANÁLISIS DE VALORES FRONTERA	3
2.2.- PRUEBA DE CAJA BLANCA	4
2.2.1.- PRUEBAS DE INSTRUCCIONES.....	4
2.2.2.- PRUEBAS DE DECISIONES	4
2.2.3.- PRUEBAS DE CONDICIONES	4
2.2.4.- PRUEBAS DE DECISIÓN/ CONDICIÓN	4
2.2.5.- PRUEBAS DE CICLOS	5
2.3.- DEBUGGING	5
2.3.1.- GUÍAS PARA DEBUGGING.....	5
3.- ESTRATEGIA DE APLICACIÓN DE LAS PRUEBAS	6
3.1.- PRUEBA DE UNIDAD.....	6
3.2.- PRUEBAS DE INTEGRACIÓN	6
3.3.- PRUEBA DE VALIDACIÓN	6
3.4.- PRUEBA DEL SISTEMA.....	6
3.5.- PRUEBA DE ACEPTACIÓN	6

GRUPO DE TRABAJO

MATERIAL_DOCENTE_ADA

*(Recopilación y elaboración de material docente para el
módulo M2 del CFGS DAI (I))*

Mª Pilar Meliό González

Beatriz Pérez Oñate

1.- INTRODUCCIÓN

“LA PRUEBA DEL SOFTWARE ES UN ELEMENTO CRÍTICO PARA LA GARANTÍA DE CALIDAD DEL SOFTWARE Y REPRESENTA UNA REVISIÓN FINAL DE LAS ESPECIFICACIONES, DEL DISEÑO Y DE LA CODIFICACIÓN”.

1.1.- OBJETIVOS DE LA PRUEBA

- ↪ La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
- ↪ Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces
- ↪ Una prueba tiene éxito si descubre un error no detectado hasta entonces.

La prueba no puede asegurar la ausencia de defectos, sólo puede demostrar que existen defectos en el software..

1.2.- ATRIBUTOS DE UNA BUENA PRUEBA

- ↪ Una buena prueba tiene una alta probabilidad de encontrar un error
- ↪ Una buena prueba no debe ser redundante.
- ↪ Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja.

2.- TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Un **CASO DE PRUEBA** es un conjunto de datos que el sistema procesará como entrada normal. Sin embargo, los datos se crean con la intención expresa de determinar si el sistema los procesará correctamente.

Por ejemplo: los casos de prueba para el manejo de inventarios deben incluir situaciones en las que las cantidades que se retiran del inventario excedan, igualen y sean menores que las existencias.

Cada caso de prueba se diseña con la intención de encontrar errores en la forma en que el sistema los procesará.

El diseño de pruebas para el software puede requerir tanto esfuerzo como el propio diseño inicial del producto.

Se debe diseñar pruebas que tengan la mayor probabilidad de encontrar de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

Cualquier producto de ingeniería puede comprobarse de una de estas dos formas:

- 1) Con un **ENFOQUE ESTRUCTURAL O PRUEBA 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) Con un **ENFOQUE FUNCIONAL O PRUEBA 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 consistirá en probar todas las posibles entradas y salidas del programa.

2.1.- PRUEBA DE CAJA NEGRA

Las pruebas de caja negra se centran en los requisitos funcionales del software.

Permiten al Ingeniero de Software obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa.

La prueba de caja negra intenta encontrar errores de las siguientes categorías:

- 1) Funciones incorrectas o ausentes
- 2) Errores de interfaz
- 3) Errores en estructuras de datos o en accesos a bases de datos externas.
- 4) Errores de rendimiento
- 5) Errores de inicialización y de terminación

2.1.1.- PARTICIÓN EQUIVALENTE

Puesto que el número de entradas que podemos probar es inmenso, se deben dividir en **CLASES EQUIVALENTES**. Estas clases están divididas de tal manera que un error por un miembro de la clase es altamente probable que haya sido causado por cualquier miembro de la clase.

Proceso:

1. Se identificarán las condiciones de las entradas.
2. A partir de ellas se identifican las posibles clases de equivalencia.
3. Las clases equivalentes deben ser creadas para encontrar entradas **válidas e inválidas**.

Reglas que ayudan a identificar clases:

Condiciones/ Especificaciones de entrada	Clases de entradas VÁLIDAS	Clases de entradas NO VÁLIDAS
Se especifica un RANGO . Ej.: El n° estará comprendido entre 1 y 49	1 clase válida: $1 \leq n^\circ \leq 49$	2 clases no válidas: $n^\circ < 1$ $n^\circ > 49$
Especificación de tipo BOOLEANO . Ej.: El primer carácter debe ser una letra	1 clase válida: "es una letra"	1 clase no válida: "no es una letra"
N° DE VALORES . Ej.: 10 caracteres	1 clase válida: 10 caracteres	2 clases no válidas: < 10 caracteres >10 caracteres
Se especifica un conjunto de VALORES ADMITIDOS . Ej.: Pueden registrarse 3 tipos de inmuebles: pisos, chalés y locales comerciales	1 clase válida por cada valor: piso chalé local comercial	1 clase no válida: Cualquier otro caso. Ej.: garaje

Una vez obtenidas las clases de equivalencia, se utilizarán para identificar los casos de prueba correspondientes:

1. Se asigna un n° único a cada clase.
2. Hasta que todas las clases válidas hayan sido cubiertas, se tratará de crear un caso de prueba que cubra tantas clases válidas como sea posible.
3. Hasta que todas las clases no válidas hayan sido cubiertas, se escribirá un caso de prueba por cada clase no válida sin cubrir.

2.1.2.- ANÁLISIS DE VALORES FRONTERA

- ↳ Por lo general, los errores ocurren en los límites de las clases equivalentes.
- ↳ Podemos fortalecer nuestras pruebas de caja negra probando los valores que se encuentran en los límites de las clases.
- ↳ En algunos casos, esto significa una prueba adicional.

Reglas:

Especificaciones	Casos a probar
RANGO DE VALORES Ej.: Entre 1.0 y 15.0	Límites y valores próximos (no válidos) Probar: 0.9, 1.0 15.0 y 15.1
N° DE VALORES: Un módulo lee una serie de valores, digamos, 1, 2 o 3 valores	probar con 1 y 3 valores

Buscar casos de prueba también para comprobar las especificaciones de rangos de las salidas.

2.2.- PRUEBA DE CAJA BLANCA

La prueba de caja blanca, denominada también prueba de caja de cristal es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para los casos de prueba. Permite obtener casos de prueba que:

- a) Garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo.
- b) Ejerciten todas las decisiones lógicas en sus vértices verdadero y falso.
- c) Ejecuten todos los lazos en sus límites y con sus límites operacionales
- d) Ejerciten las estructuras internas de datos para asegurar su validez.

2.2.1.- PRUEBAS DE INSTRUCCIONES

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

2.2.2.- PRUEBAS 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.

2.2.3.- PRUEBAS 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.

2.2.4.- PRUEBAS DE DECISIÓN/ CONDICIÓN

Más potentes, consisten en exigir el criterio de cobertura de condiciones y el de decisiones también. Ej: (a>0) and (op='N')

A=0 , op='N'	}	Son prueba de condiciones pero no de decision.
a=1 , op= 'S'		

2.2.5.- PRUEBAS DE CICLOS

Mientras las pruebas de decisiones cubren parcialmente los ciclos, las pruebas de ciclos es más completo y se asegura que cada ciclo sea ejecutado para:

- cero iteraciones
- una interacción
- muchas iteraciones

Ejemplos:

while I < 10 do begin I = I + 1; end;	for I := 1 to N do	repeat I := I - 1 until I < N;
cero I = 10	cero N = 0	cero = imposible
una I = 9	una I = 1	una I = N
muchas I = 4	muchas I = 12	muchas I = N + 10

2.3.- DEBUGGING

Debugging es el proceso que toma parte después de las pruebas. Las pruebas muestran que existe un error, debugging entonces toma parte y tiene dos faSes:

- localizar donde ocurrió el error
- corregir el error

Debugging requiere pensar, por lo que es difícil.

2.3.1.- GUÍAS PARA DEBUGGING

Empezar desde el error

- trabaje hacia atrás explicando que ha pasado

Empezar desde el último output correcto

- trabaje hacia delante formando una explicación hasta el error

Explique su problema a alguien

- Comentar el problema muchas veces lleva a la solución sin que la otra persona hable y por último,

Don't Panic: nunca haga cambios aleatorios

3.- ESTRATEGIA DE APLICACIÓN DE LAS PRUEBAS

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.

3.1.- PRUEBA DE UNIDAD

Centra el proceso en la verificación de la menor unidad del diseño del software: el módulo. Especialmente orientado a la prueba de la caja blanca aunque se completa también con la prueba de la caja negra.

3.2.- PRUEBAS DE INTEGRACIÓN

Los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto. Ha de tener en cuenta las interfaces entre componentes de la arquitectura del software.

3.3.- PRUEBA DE VALIDACIÓN

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.

3.4.- PRUEBA DEL SISTEMA


El software ya validado se integra con el resto del sistema (Ej.: elementos mecánicos, interfaces electrónicas..) para probar su funcionamiento conjunto. Se efectúan además sobre procesos que afectan a la seguridad en el acceso a los datos, rendimiento bajo condiciones de grandes cargas de trabajo, tolerancia a fallos y recuperación del sistema.

3.5.- PRUEBA DE ACEPTACIÓN

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.

BIBLIOGRAFÍA:

 *DSIC EUI-FI UPV Transparencias*

 *“Metodología y tecnología de la programación”*
Antonio Molina , Patricio Letelier, Pedro Torres, Juan Sánchez.
UPV. Servicio de publicaciones