TEMA III
DISEÑO Y REALIZACIÓN DE PRUEBAS

Planificación de las pruebas

Durante todo el proceso de desarrollo de software, desde la fase de diseño, en la implementación y una vez desarrollada la aplicación, es necesario realizar un conjunto de pruebas que permitan verificar que el software que se está creando, es correcto y cumple con las especificaciones impuesta por el usuario.

Las pruebas son el proceso que permite verificar y revelar la calidad de un producto software. Se utilizan para identificar posibles fallos de implementación, calidad o usabilidad de un programa.

Para probar un programa hay que someterlo a todas las posibles variaciones de los datos de entrada, tanto si son válidos como si no lo son, y ver el resultado obtenido.
Un plan de pruebas o juego de ensayo es un conjunto de pruebas en las que se explican detalladamente los datos de entrada, el resultado que se espera obtener y el resultado obtenido.

Una buena prueba debe centrarse en dos objetivos:

- Probar si el software no hace lo que debe hacer
- Probar si el software hace lo que no debe hacer.

En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades donde es muy fácil que se produzca un error humano. Estos errores humanos pueden ser:

- Una incorrecta especificación de los objetivos,
- Errores producidos durante el proceso de diseño
- Errores que aparecen en la fase de desarrollo.

Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas:

a) Empezarían con la **prueba de unidad**, donde se analizaría el código implementado

b) Seguiríamos con la **prueba de integración,** donde se prestan atención al diseño y la construcción de la arquitectura del software.

c) El siguiente paso sería la **prueba de validación,** donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software.

d) Finalmente se alcanza la **prueba de sistema** que verifica el funcionamiento total del software y otros elementos del sistema.

Tipos de pruebas

- **Funcionales:** Evalúan el cumplimiento de los requisitos.
- **No funcionales**: Evalúan aspectos adicionales como rendimiento, seguridad, ...

Pruebas unitarias.

Las pruebas unitarias, o prueba de la unidad, tienen por objetivo probar el correcto funcionamiento de un módulo de código. El fin que se persigue, es que cada módulo funciona correctamente por separado. Posteriormente, con la prueba de integración, se podrá asegurar el correcto funcionamiento del sistema.

Una unidad es la parte de la aplicación más pequeña que se puede probar.

- En programación procedural, una unidad puede ser una función o procedimiento,
- En programación orientada a objetos, una unidad es normalmente un método.

En el diseño de los casos de pruebas unitarias, habrá que tener en cuenta los siguientes requisitos:

- Automatizable: no debería requerirse una intervención manual.
- Completas: deben cubrir la mayor cantidad de código.
- Repetibles o Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra.
- Profesionales: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

El objetivo de las pruebas unitarias es aislar cada parte del programa y demostrar que las partes individuales son correctas. Las pruebas individuales nos proporcionan cinco ventajas básicas:

1. Fomentan el cambio: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.

2. Simplifica la integración: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.

3. Documenta el código: Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.

4. Separación de la interfaz y la implementación: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.

5. Los errores están más acotados y son más fáciles de localizar: dado que tenemos pruebas unitarias que pueden desenmascararlos.

Pruebas de Integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema, involucran a varios módulos y terminan probando el sistema como conjunto.

Las pruebas estructurales de integración son similares a las pruebas de caja blanca; nos referiremos a llamadas entre módulos, se trata de identificar todos los posibles esquemas de llamadas y ejercitarlos para lograr una buena cobertura.

Las pruebas funcionales de integración son similares a las pruebas de caja negra. Aquí trataremos de encontrar fallos en la respuesta de un módulo cuando su operación depende de los servicios prestados por otro módulo.

Las pruebas finales de integración cubren todo el sistema y pretenden cubrir plenamente la especificación de requisitos del usuario.

JUnit

Junit es un Framework Open Source para la automatización de las pruebas (tanto unitarias, como de integración) en los proyectos Software. El framework provee al usuario de herramientas, clases y métodos que le facilitan la tarea de realizar pruebas en su sistema y así asegurar su consistencia y funcionalidad.

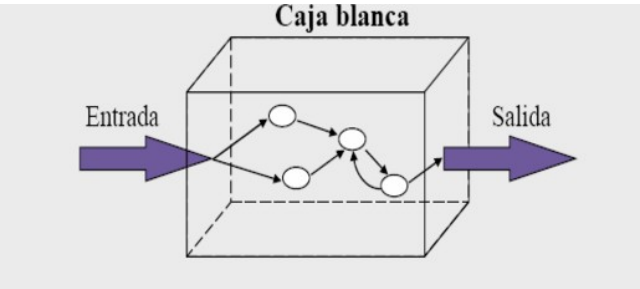
Estrategias de prueba

En la ingeniería del software nos encontramos dos enfoques fundamentalmente:

1. Prueba de la Caja Negra (Black Box Testing): cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación son los esperados en función de las entradas que recibe.

Una prueba de tipo Caja Negra se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento interno del sistema. Cuando se realiza este tipo de pruebas, solo se conocen las entradas adecuadas que deberá recibir la aplicación, así como las salidas que les correspondan, pero no se conoce el proceso mediante el cual la aplicación obtiene esos resultados.

2. Prueba de la Caja Blanca (White Box Testing): en este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.



En contraposición a lo anterior, una prueba de Caja Blanca va a analizar y probar directamente el código de la aplicación. Para llevar a cabo una prueba de Caja Blanca, es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas.

Pruebas de caja negra o funcionales

También se las conoce como

1. Pruebas de caja opaca
2. Pruebas funcionales
3. Pruebas de entrada/salida
4. Pruebas inducidas por los datos

Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.

Si, por ejemplo, estamos implementando una aplicación que realiza un determinado cálculo científico, en este enfoque sólo nos interesa verificar que, para una determinada entrada a ese programa, el resultado de la ejecución devuelve los datos esperados. Nunca consideraríamos el código desarrollado, el algoritmo utilizado, la eficiencia, si hay partes de código innecesarias….

Las pruebas de caja negra están especialmente indicadas en aquellos módulos que van a interactuar con el usuario.

Se sigue una técnica conocida como "***clases de equivalencia***", que consiste en probar sólo un caso de cada clase, ya que los demás datos de la misma clase son equivalentes.

Así se identifican las clases de equivalencia:

- Si un dato de entrada debe estar comprendido en un cierto rango, aparecen 3 clases de equivalencia: por debajo, en el rango y por encima del rango.
- Si un dato de entrada requiere un valor concreto, aparecen 2 clases de equivalencia: el valor concreto y otro valor distinto de éste.
- Si un dato de entrada requiere un valor de entre los de un conjunto, aparecen 2 clases de equivalencia: en el conjunto o fuera de él.
- Si una entrada es booleana, hay 2 clases: si o no.

Ejemplo: Programa que lee un número de día de la semana y nos dice si es lunes, martes, miércoles... Así, hay 3 clases:

1. Números menores que 1
2. Números entre 1 y 7
3. Números mayores que 7

Al leer los requisitos del programa, nos podemos encontrar con una serie de valores singulares, que marcan diferencias de comportamiento. Estos valores son claros candidatos a marcar clases de equivalencia: por abajo y por arriba. A estos valores se les denomina "frontera" o **valores límite.**

Limitaciones

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable; pero no suficiente. Un programa puede pasar millones de pruebas y sin embargo tener defectos internos que surgen en el momento más inoportuno.

Por ejemplo, un PC que contenga el virus viernes-13 puede estar pasando pruebas de caja negra durante años y años. Sólo falla si es viernes y es día 13; pero ¿a quién se le iba a ocurrir hacer esa prueba?

Las pruebas de caja negra nos convencen de que un programa hace lo que queremos; pero no de que haga (además) otras cosas menos aceptables

Ejemplo: Dada esta función
Las clases de equivalencia serían:

```
public double funcion1 (double x)
{
    if (x > 0 && x < 100)
        return x+2;
    else
        return x-2;
}
```

1. Por debajo: x<=0
2. En: x>0 y x< 100
3. Por encima: x>=100

CONDICIONES	CLASES VÁLIDAS	CLASES NO VÁLIDAS
Número entero	>0 y <100 (1)	<=0 (2)
		>=100 (3)

Los casos de prueba podrían ser:

1. Por debajo: x=0, x=-2
2. En: x=50
3. Por encima: x=100, x=200

CLASES	COBERTURA	VALOR LÍMITE	ENTRADA	SALIDA OBTENIDA	SALIDA ESPERADA
VÁLIDAS	(1)	1	1		
		99	99		
			50		
NO VÁLIDAS	(2)	0	0		
			-2		
	(3)	100	100		
			200		

Ejemplo

En el código Java adjunto, aparecen dos funciones que reciben el parámetro x. En la función1, el parámetro es de tipo real y en la función2, **el** parámetro es de tipo entero.

```
public double funcion1 (double x)
{
    if (x > 5)
        return x;
    else
        return -1;
}

public int funcion2 (int x)
{
    if (x > 5)
        return x;
    else
        return -1;
}
```

El código de las dos funciones es el mismo, sin embargo, los casos de prueba con valores límite va a ser diferente.
Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.

Cuando hay que seleccionar un valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.

Si el parámetro x de entrada tiene que ser mayor estricto que 5, y el valor es real, los valores límite pueden ser 4,99, 5,00 y 5,01.
Si el parámetro de entrada x tiene que ser mayor estricto que 5, y el valor es entero, los valores límite pueden ser 4,5,6

Pruebas de caja blanca o estructurales

Son las pruebas de caja blanca. Con este tipo de pruebas, se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo. Este tipo de pruebas no pretenden comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas las instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer… Su objetivo es probar exhaustivamente la estructura del código.

Se llama cobertura, al porcentaje de código que ha sido probado o “cubierto” con las pruebas.

Hay varios tipos:

- **Cobertura de segmentos**: Se escriben casos de prueba para que cada sentencia (sin condición) sea probada al menos una vez.
- **Cobertura de ramas**: Se escriben casos de prueba suficientes para que cada decisión se ejecute una vez con resultado positivo y otra con negativo.

  Ej. if (a>0)

  b++;

  Se tendría que probar por ejemplo a=5 y a=-2

- **Cobertura de condición/decisión:** trocea las expresiones booleanas complejas en sus componentes e intenta cubrir todos los posibles valores de cada uno de ellos.

  Ej. if (a>0 || b<0) a++;

  else a--;

  Se tendrían que probar 4 casos;

  a=5 b=3

  a=5 b=-3

  a=-5 b=3

  a=-5 b=-3

- **Cobertura de bucles:** Se trata de probar que se entra en los bucles 0, 1 o más veces (1 o más veces en el do-while), así como las salidas por break.

Ejemplo

Dada esta función que forma parte de una programación

```
int prueba (int x, int y)
{
        int z=0;
        if ((x>0) && (y>0))
        {
                z=x;
        }
        return z;
}
```

Si durante la ejecución del programa, la función es llamada, al menos una vez, el cubrimiento de la función es satisfecho.

El cubrimiento de sentencias para esta función será satisfecho si es invocada, por ejemplo, con caso de prueba (1,1), ya que, en este caso, cada línea de la función se ejecuta

Si invocamos a la función con prueba (1,1) y prueba (0,1), se satisfará el cubrimiento de rama.

El cubrimiento de condición puede satisfacerse si probamos con prueba (1,1), prueba (1,0) prueba (0,0) y prueba (0,1).

En la práctica, se suelen aplicar las pruebas de caja negra, y a aquellos tramos de código por los que no se ha pasado se les aplican pruebas de caja blanca, para llegar a alcanzar una cobertura cercana al 100%. Esta cobertura dependerá del tipo de programa (si es un juego, bastará el 80%, si es una aplicación médica el 99%, etc).

Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace; pero no de que haga lo que necesitamos.

Tipos de prueba.

No existe una clasificación oficial o formal sobre los diversos tipos de pruebas de software. En la ingeniería del software nos encontramos con los siguientes tipos de pruebas

	Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados
Prueba estructural	El enfoque estructural o de caja blanca se centra en la estructura interna del programa (analiza los caminos de ejecución).
Prueba funcional	El enfoque funcional o de caja negra se centra en las funciones, entradas y salidas que recibe y produce un módulo o función concreta.
Pruebas aleatorias	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.
Pruebas de regresión	Son cualquier tipo de pruebas de software que intentan descubrir las causas de nuevos errores, carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error. Esto implica que el error tratado se produce como consecuencia inesperada del citado cambio en el programa.

Tipos de pruebas de software	
Pruebas de unidad	Con ella se va a probar el correcto funcionamiento de un módulo de código
Pruebas de carga	Este es el tipo más sencillo de pruebas de rendimiento. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada. Esta carga puede ser el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga. Esta prueba puede mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si la base de datos, el servidor de aplicaciones, etc también se monitorizan, entonces esta prueba puede mostrar el cuello de botella en la aplicación.
Prueba de estrés	Esta prueba se utiliza normalmente para romper la aplicación. Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.
Prueba de estabilidad	Esta prueba normalmente se hace para determinar si la aplicación puede aguantar una carga esperada continuada. Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación
Pruebas de picos	La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan, como cuando tiene cambios drásticos en su carga.

Pruebas de validación

En el proceso de validación, interviene de manera decisiva el cliente. Hay que tener en cuenta, que estamos desarrollando una aplicación para terceros, y que son estos los que deciden si la aplicación se ajusta a los requerimientos establecidos en el análisis.

En la validación intentan descubrir errores, pero desde el punto de vista de los requisitos (Comportamiento y casos de uso que se esperan que cumpla el software que se está diseñando).

La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Un plan de prueba traza la clase de pruebas que se han de llevar a cabo, y un procedimiento de prueba define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos.

Tanto el plan como el procedimiento estarán diseñados para asegurar que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligible y que se alcanzan otros requisitos, como

portabilidad (Capacidad de un programa para ser ejecutado en cualquier arquitectura física de un equipo), compatibilidad, recuperación de errores, facilidad de mantenimiento etc.

Cuando se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes:

- Las características de funcionamiento o rendimiento están de acuerdo con las especificaciones y son aceptables

- Se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada

Otros tipos de pruebas

Recorridos (walkthroughs)

Consiste en sentar alrededor de una mesa a los desarrolladores y a una serie de críticos, bajo las órdenes de un moderador que impida un recalentamiento de los ánimos. El método consiste en que los revisores se leen el programa y piden explicaciones de todo lo que no está meridianamente claro. Puede que simplemente falte un comentario explicativo, o que detecten un error auténtico o que simplemente el código sea tan complejo de entender/explicar que más vale que se rehaga de forma más simple. Para un sistema complejo pueden hacer falta muchas sesiones.

Esta técnica es muy eficaz localizando errores de naturaleza local; pero falla estrepitosamente cuando el error deriva de la interacción entre dos partes alejadas del programa. Nótese que no se está ejecutando el programa, sólo mirándolo con lupa, y de esta forma sólo se ve en cada instante un trocito del listado.

Aleatorias (random testing)

Realizar la fase de pruebas con una serie de casos elegidos al azar. Esto pondrá de manifiesto los errores más patentes. No obstante, pueden permanecer ocultos errores más sibilinos que sólo se muestran ante entradas muy precisas.

Si el programa es poco crítico (una aplicación personal, un juego, ...) puede que esto sea suficiente. Pero si se trata de una aplicación militar o con riesgo para vidas humanas, es de todo punto insuficiente.

Solidez (robustness testing)

Se prueba la capacidad del sistema para salir de situaciones embarazosas provocadas por errores en el suministro de datos. Estas pruebas son importantes en sistemas con una interfaz al exterior, en particular cuando la interfaz es humana.

Por ejemplo, en un sistema que admite una serie de órdenes (commands) se deben probar los siguientes extremos:

- Órdenes correctas, todas y cada una
- Órdenes con defectos de sintaxis, tanto pequeñas desviaciones como errores de bulto
- Órdenes correctas, pero en orden incorrecto, o fuera de lugar la orden nula (línea vacía, una o más)
- Órdenes correctas, pero con datos de más provocar una interrupción (BREAK, ^C, o lo que corresponda al sistema soporte) justo después de introducir una orden.
- Órdenes con delimitadores inapropiados (comas, puntos, ...)
- Órdenes con delimitadores incongruentes consigo mismos (por ejemplo, esto]

Aguante (stress testing)

En ciertos sistemas es conveniente saber hasta dónde aguantan, bien por razones internas (¿hasta cuantos datos podrá procesar?), bien externas (¿es capaz de trabajar con un disco al 90%?, ¿aguanta una carga de la CPU del 90?, etc etc)

Prestaciones (performance testing)

A veces es importante el tiempo de respuesta, u otros parámetros de gasto. Típicamente nos puede preocupar cuánto tiempo le lleva al sistema procesar tantos datos, o cuánta memoria consume, o cuánto espacio en disco utiliza, o cuántos datos transfiere por un canal de comunicaciones, o ... Para todos estos parámetros suele ser importante conocer cómo evolucionan al variar la dimensión del problema (por ejemplo, al duplicarse el volumen de datos de entrada).

Conformidad u Homologación (conformance testing)

En programas de comunicaciones es muy frecuente que, además de los requisitos específicos del programa que estamos construyendo, aparezca alguna norma más amplia a la que el programa deba atenerse. Es frecuente que organismos internacionales como ISO y el CCITT elaboren especificaciones de referencia a las que los diversos fabricantes deben atenerse para que sus ordenadores sean capaces de entenderse entre sí.

Las pruebas, de caja negra, que se le pasan a un producto para detectar discrepancias respecto a una norma de las descritas en el párrafo anterior se denominan de conformidad u homologación. Suelen realizarse en un centro especialmente acreditado al efecto y, si se pasan satisfactoriamente, el producto recibe un sello oficial que dice: "homologado".

Interoperabilidad (interoperability tesing)

En el mismo escenario del punto anterior, programas de comunicaciones que deden permitir que dos ordenadores se entiendan, aparte de las pruebas de conformidad se suelen correr una serie de pruebas, también de caja negra, que involucran 2 o más productos, y buscan problemas de comunicación entre ellos.

Regresión (regression testing)

Todos los sistemas sufren una evolución a lo largo de su vida activa. En cada nueva versión se supone que o bien se corrigen defectos, o se añaden nuevas funciones, o ambas cosas. En cualquier caso, una nueva versión exige una nueva pasada por las pruebas. Si éstas se han sistematizado en una fase anterior, ahora pueden volver a pasarse automáticamente, simplemente para comprobar que las modificaciones no provocan errores donde antes no los había.

El mínimo necesario para usar unas pruebas en una futura revisión del programa es una documentación muy clara.

CONCLUSIONES

1. Probar es ejecutar un programa para encontrarle fallos. Jamás se debería probar un programa con el ánimo de mostrar que funciona; ese no es el objetivo.
2. Un caso de prueba tiene éxito cuando encuentra un fallo.
3. Las pruebas debe diseñarlas y pasarlas una persona distinta de la que ha escrito el código; es la única forma de no ser "comprensivo con los fallos".
4. Las pruebas hay que empezar a pasarlas antes de terminar el programa.
5. Si en un módulo se encuentran muchos fallos, hay que insistir sobre él, y a veces es más rentable volver a reescribirlo.
6. Las pruebas pueden encontrar fallos; pero jamás demostrar que no los hay.
7. Las pruebas también tienen fallos

Herramientas de Depuración

Una vez que advertimos que nuestro código no pasa las pruebas, podemos utilizar el depurador (Modo Debug) del IDE que empleemos para revisar el estado de nuestro programa mientas lo vamos ejecutando paso a paso y corregir los posibles errores que pueda contener.
Es recomendable establecer breakpoints en lugares cercanos al módulo o parte del código que queremos evaluar e investigar.