Unidad 3. Diseño y realización de pruebas

Entornos de desarrollo
1º Desarrollo de Aplicaciones Multiplataforma

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**

Permiten verificar que el software que se está **creando**, es **correcto** y cumple con las **especificaciones** del usuario

Planificación de las pruebas

En el proceso de desarrollo de software es muy fácil que se produzca un **error humano**

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

Planificación de las pruebas

Mediante la realización de **pruebas de software**, se van a realizar las tareas de verificación y validación del software

La **verificación** es la comprobación que un sistema o parte de un sistema, cumple con las **condiciones impuestas.** Se comprueba si la aplicación **se está construyendo correctamente**

La **validación** es el proceso de evaluación del sistema o de uno de sus componentes, para determinar si **satisface los requisitos especificados**

Planificación de las pruebas

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

Siguiendo el **Modelo en Espiral**:

1. La prueba de **unidad**: se analizaría el código implementado
2. La prueba de **integración**: se prestan atención al diseño y la construcción de la arquitectura del software
3. La prueba de **validación**: se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software
4. La prueba de **sistema**: verifica el funcionamiento total del software y otros elementos del sistema

Tipos de prueba

Prueba de la Caja Negra (Black Box Testing): es probada usando su **interfaz externa**

Comprueba que los r**esultados de la ejecución** de la aplicación son los esperados, en función de las entradas que recibe

No es necesario conocer la **estructura**, ni el funcionamiento interno del sistema

Sólo se conocen las **entradas** adecuadas que deberá recibir la aplicación y las **salidas** que les correspondan

Tipos de prueba

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

Analiza y prueba directamente el **código de la aplicación**

Tipos de prueba

Pruebas de unidad: se va a probar el correcto funcionamiento de un módulo de código

Pruebas de carga: Se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada

- **Número esperado de usuarios concurrentes** utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga
- Puede mostrar los **tiempos de respuesta** de todas las transacciones importantes de la aplicación

Tipos de prueba

Prueba de estrés: 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
- 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

Tipos de prueba

Prueba de estabilidad: se hace para determinar si la aplicación puede aguantar una **carga esperada continuada**

Pruebas de picos: trata de observar el comportamiento del sistema **variando el número de usuarios**, tanto cuando bajan, como cuando tiene cambios drásticos en su carga

Tipos de prueba

Pruebas aleatorias: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: intentan descubrir las causas de nuevos errores, carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software inducidos por **cambios recientes** realizados en partes de la aplicación

Reflexión

Resulta habitual, que en una empresa de desarrollo de software se gaste el **40 por ciento del esfuerzo de desarrollo en la prueba**

¿Por qué es tan importante la prueba?

¿Qué tipos de errores se intentan solucionar con las pruebas?

Reflexión

Las pruebas son muy importantes, ya que **permiten descubrir errores**, **fallos en la implementación**, **calidad** o **usabilidad** del software, ayudando a **garantizar la calidad**

Con las pruebas se intenta verificar que cada componente que se ha diseñado, ya sea un método, función, módulo, etc. **realiza la función para la que se ha diseñado**. También se intenta comprobar, que existen condiciones en las que todos los caminos de una aplicación llegan a ejecutarse

Prueba funcionales

Prueba de la Caja Negra (Black Box Testing)

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, sólo si realiza las funciones que se esperan de él

Prueba funcionales

Comprenderán aquellas actividades cuyo objetivo sea **verificar una acción específica o funcional** dentro del código de una aplicación

Las pruebas funcionales intentarían responder a las preguntas

- ¿puede el usuario hacer esto?
- ¿funciona esta utilidad de la aplicación?

Prueba funcionales

Principal cometido: **comprobar el correcto funcionamiento de los componentes de la aplicación informática**

Para realizar este tipo de pruebas, se deben analizar las **entradas** y las **salidas** de cada componente, verificando que el **resultado es el esperado**

Solo se van a considerar las entradas y salidas del sistema, **sin preocuparnos por la estructura interna del mismo**

Prueba funcionales

Ejemplo: estamos implementando una a**plicación que realiza un determinado cálculo científico**, en el enfoque de las pruebas funcionales, solo nos interesa verificar que **ante una determinada entrada** a ese programa el resultado de la ejecución del mismo **devuelve como resultado los datos esperados**

Este tipo de prueba, **no consideraría, en ningún caso, el código desarrollado**, ni el algoritmo, ni la eficiencia, ni si hay partes del código innecesarias, etc.

Prueba funcionales

Tres tipos de pruebas:

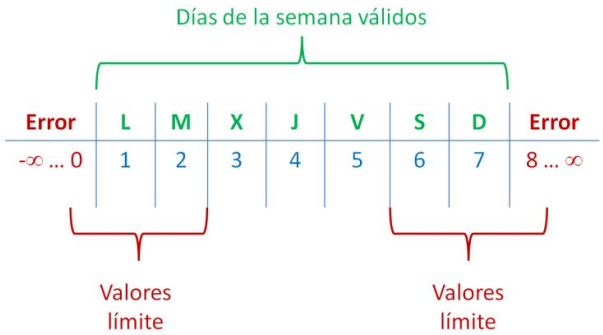
- **Particiones equivalentes**: considera el menor **número posible de casos de pruebas**, para ello, cada caso de prueba tiene que abarcar el **mayor número posible de entradas diferentes**

  Lo que se pretende, es crear un conjunto de **clases de equivalencia**, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.

Prueba funcionales

Tres tipos de pruebas:

- **Análisis de valores límite:** se eligen como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia



Prueba funcionales

Tres tipos de pruebas:

- **Pruebas aleatorias:** generar entradas aleatorias para la aplicación que hay que probar

  Se suelen utilizar **generadores de prueba**, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación

  Esta tipo de pruebas, se suelen utilizar en **aplicaciones que no sean interactivas**, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos.

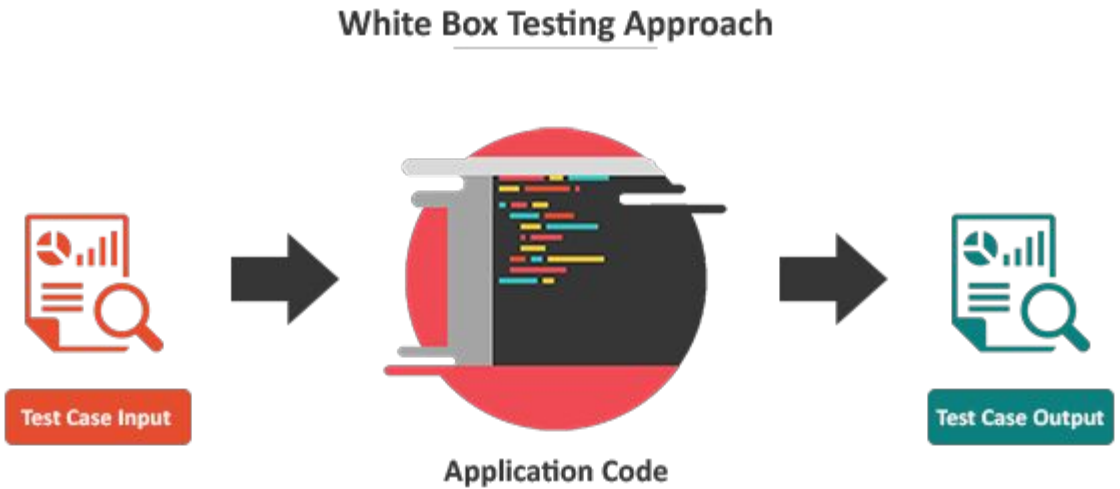
Prueba estructurales

Prueba de la Caja Blanca (White Box Testing)

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 la instrucciones** del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, etc.

Prueba estructurales

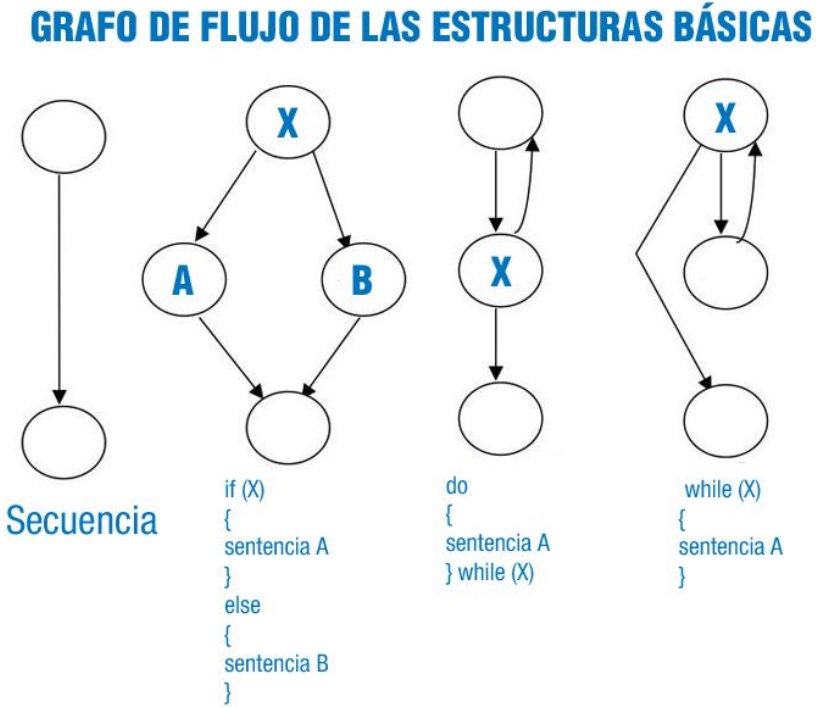


Prueba estructurales

Criterios de cobertura lógica que determina la mayor o menor seguridad en la detección de errores:

- **Cobertura de sentencias**: generar casos de pruebas suficientes para que **cada instrucción del programa sea ejecutada, al menos, una vez**
- **Cobertura de decisiones**: crear los suficientes casos de prueba para que **cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez** a cierto y otra a falso
- **Cobertura de condiciones**: crear los suficientes casos de prueba para que **cada elemento de una condición, se evalúe al menos una vez a falso y otra a verdadero**
- **Cobertura de condiciones y decisiones**: consiste en cumplir simultáneamente las dos anteriores

Prueba estructurales



Prueba estructurales

- **Cobertura de caminos:** es el criterio más importante, se debe **ejecutar al menos una vez cada secuencia de sentencias encadenadas**, desde la sentencia inicial del programa, hasta su sentencia final
  - La ejecución de este conjunto de sentencias, se conoce como **camino**
  - Como el número de caminos que puede tener una aplicación, puede ser muy grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.
- **Cobertura del camino de prueba:** dos variantes
  - una indica que **cada bucle se debe ejecutar sólo una vez**, ya que hacerlo más veces no aumenta la efectividad de la prueba
  - otra que recomienda que se pruebe **cada bucle tres veces**: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces

Prueba estructurales

En las pruebas de caja negra:

A. Es necesario conocer el código fuente del programa, para realizar las pruebas.
B. Se comprueba que todos los caminos del programa, se pueden recorrer, al menos una vez.
C. Se comprueba que los resultados de una aplicación, son los esperados para las entradas que se le han proporcionado.
D. Es incompatible con la prueba de caja blanca.

Prueba estructurales

En las pruebas de caja negra:

A. Es necesario conocer el código fuente del programa, para realizar las pruebas.
B. Se comprueba que todos los caminos del programa, se pueden recorrer, al menos una vez.
C. Se comprueba que los resultados de una aplicación, son los esperados para las entradas que se le han proporcionado.
D. Es incompatible con la prueba de caja blanca.

Prueba de regresión

Objetivo: comprobar que los **cambios** sobre un componente de una aplicación, no introduce un comportamiento no deseado o errores adicionales en otros componentes no modificados

Se deben llevar a cabo **cada vez que se hace un cambio en el sistema**, tanto para corregir un error, como para realizar una mejora

No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes

Prueba de regresión

Repetición de las pruebas que ya se hayan realizado previamente, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados

Prueba de regresión

Tres clases diferentes de clases de prueba:

- Una muestra representativa de pruebas que ejercite **todas las funciones del software**
- Pruebas adicionales que se centran en las **funciones del software que se van a ver probablemente afectadas por el cambio**
- Pruebas que se centran en los **componentes del software que han cambiado**

Prueba de regresión

La prueba de regresión:

A. Se realiza una vez finalizado cada módulo del sistema a desarrollar
B. Solo utiliza el enfoque de la caja negra
C. Se realiza cuando se produce una modificación, debido a la detección de algún error, en la fase de prueba
D. Es incompatible con la prueba de caja blanca

Prueba de regresión

La prueba de regresión:

A. Se realiza una vez finalizado cada módulo del sistema a desarrollar
B. Solo utiliza el enfoque de la caja negra
C. Se realiza cuando se produce una modificación, debido a la detección de algún error, en la fase de prueba
D. Es incompatible con la prueba de caja blanca

Actividad pruebas de aplicación

Grupos de 3 o 4 personas

Imaginad que trabajáis en el desarrollo de una aplicación:

- Definid los requisitos funcionales y los no funcionales
- Definid cómo realizaríais las pruebas comentadas en clase (funcionales, estructurales, de regresión...)

Realizad una pequeña exposición de la información anterior

Procedimientos y casos de prueba

Caso de prueba: conjunto de **entradas**, **condiciones de ejecución** y **resultados esperados**, desarrollados para un objetivo particular

Ejemplo: ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada

Procedimientos y casos de prueba

Complejidad de las aplicaciones informáticas

Prácticamente **imposible probar todas la combinaciones**

En el diseño de los casos de prueba, siempre es necesario asegurar que con ellos se obtiene un **nivel aceptable** de **probabilidad** de que se detectarán los errores existentes

Procedimientos y casos de prueba

Procedimientos para el diseño de los casos de prueba:

- **Enfoque funcional o de caja negra**: recibe un entrada de forma adecuada y se produce una salida correcta
- **Enfoque estructural o caja blanca**: comprobar que la operación interna se ajusta a las especificaciones
- **Enfoque aleatorio**: a partir de modelos obtenidos estadísticamente, se elaboran casos de prueba que prueben las entradas del programa

Herramientas de depuración

Cada **IDE incluye herramientas de depuración**

- inclusión de puntos de ruptura
- ejecución paso a paso de cada instrucción
- ejecución por procedimiento
- inspección de variables
- etc.

Herramientas de depuración

Corrección de errores lógicos (bugs) estos no evitan que el programa se pueda **compilar con éxito**, ya que no hay errores sintácticos, ni se utilizan variables no declaradas, etc.

Los **errores lógicos**, pueden provocar que el programa **devuelva resultados erróneos**, que no sean los esperados o pueden provocar que el programa **termine antes de tiempo** o **no termine nunca**

Herramientas de depuración

Depurador: permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos

Un programa **debe compilarse con éxito** para poder utilizarlo en el depurador

Nos permita **analizar** todo el programa, **mientras éste se ejecuta**

Permite **suspender la ejecución de un programa**, examinar y establecer los **valores de las variables**, comprobar los **valores devueltos** por un determinado método, el resultado de una comparación lógica o relacional, etc

Depurador: Puntos de ruptura

Breakpoint

Se selecciona la **línea de código** donde queremos que el programa se **pare**, para a partir de ella, **inspeccionar variables**, o realizar una **ejecución paso a paso**, para verificar la corrección del código

Depurador: Puntos de ruptura

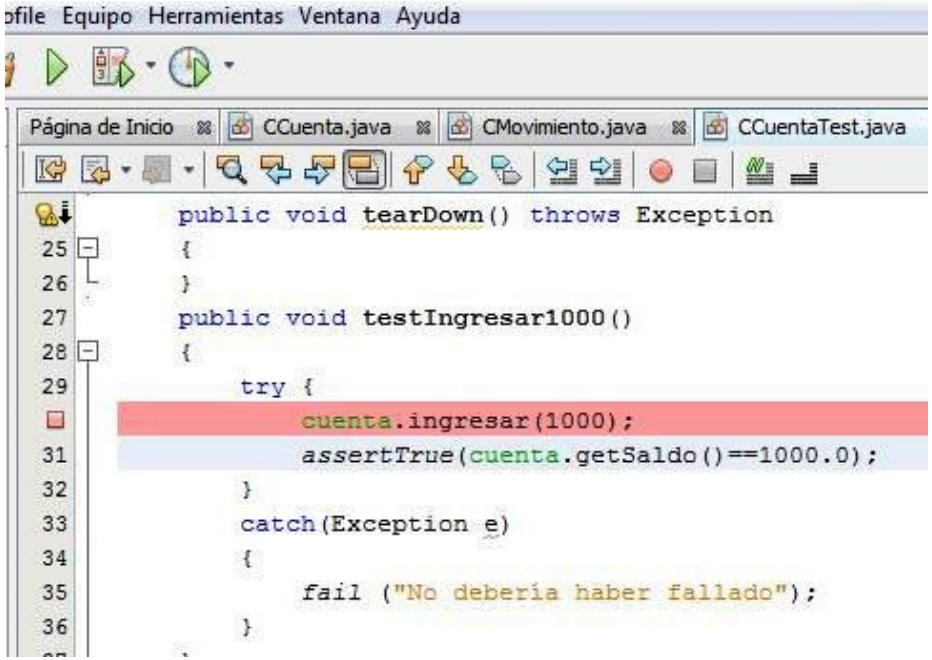
Una vez insertado el punto de ruptura, e iniciada la depuración, el programa a evaluar se ejecutaría hasta la l**ínea marcada con el punto de ruptura**

Se pueden **examinar las variables**, y comprobar que los **valores** que tienen asignados son correctos,

Se pueden iniciar una depuración **paso a paso**, e ir comprobando el camino que toma el programa a partir del punto de ruptura

Una vez realiza la comprobación, podemos **abortar el programa**, o **continuar la ejecución** normal del mismo

Depurador: Puntos de ruptura



Se pueden insertar varios puntos de ruptura

Depurador: Tipos de ejecución

Paso a paso por instrucción

Paso a paso por procedimiento

Ejecución hasta una instrucción

Ejecución de un programa hasta el final del programa

Depurador: Tipos de ejecución

Paso a paso por instrucción

Línea por línea, para buscar y corregir errores lógicos

El avance paso a paso a lo largo de una parte del programa puede ayudarnos a verificar que el código de un método se ejecute en forma correcta

Depurador: Tipos de ejecución

Paso a paso por procedimiento

Permite introducir los parámetro que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado

Cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interese volver a depurarlo, sólo nos interesa el valor que devuelve

Depurador: Tipos de ejecución

Ejecución hasta una instrucción

El depurador ejecuta el programa, y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto, podemos hacer una depuración paso a paso o por procedimiento

Depurador: Tipos de ejecución

Ejecución de un programa hasta el final del programa

Ejecutamos las instrucciones de un programa hasta el final, sin detenernos en las instrucciones intermedias

Depurador: Tipos de ejecución

Los distintos modos de ejecución, se van a ajustar a las **necesidades de depuración** que tengamos en cada momento

Si hemos probada un método, y sabemos que funciona correctamente, no es necesario realizar una ejecución paso a paso en él

Depurador: Examinadores de variables

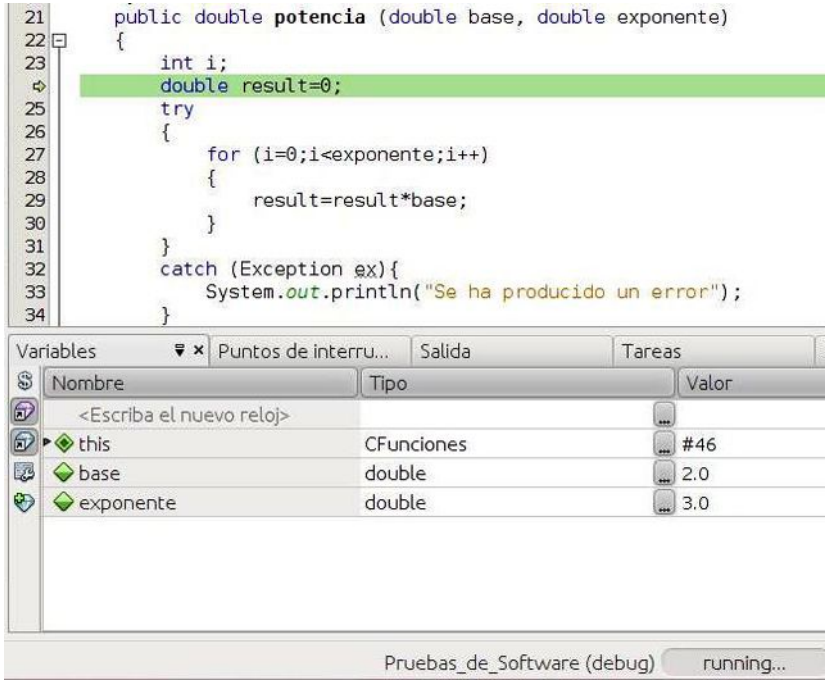
Comprobar que la aplicación funciona de manera adecuada

Variables vayan tomando los **valores adecuados** en cada momento

Con los examinadores de variables, podemos comprobar los distintos valores que adquiere las variables, así como su tipo

NetBeans -> **Ventana de Inspección**. **Agregar** aquellas **variables** de las que tengamos interés en inspeccionar su valor. Conforme el programa **se vaya ejecutando**, NetBeans irá mostrando los valores que toman las variables

Depurador: Examinadores de variables



Depurador: Examinadores de variables

En el ejemplo anterior:

- El programa llega a una función de nombre **potencia**
- Esta función tiene definida **tres variables**
- A lo largo de la ejecución del bucle, vemos como la variable result, van cambiando de valor
- **Error** si con valores de entrada para los que conocemos el resultado, la función **no devuelve el valor esperado**

Validaciones

Descubrir **errores desde el punto de vista de los requisitos**

Se consigue mediante pruebas de **caja negra** que demuestran la **conformidad con los requisitos**

Plan de prueba: traza la clase de pruebas que se han de llevar a cabo

Procedimiento de prueba: define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos

Validaciones

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**, **compatibilidad**, recuperación de **errores**, facilidad de **mantenimiento** etc.

Validaciones

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**

Pruebas de código

Consiste en la **ejecución de un programa con el objetivo de encontrar errores**

El programa o parte de él, se va a **ejecutar bajo unas condiciones previamente especificadas**, para una vez observados los resultados, estos sean registrados y evaluados

Pruebas de código: cubrimiento

Realizado por el programador o programadora y consiste en **comprobar que los caminos definidos en el código se pueden llegar a recorrer**

Tipo de prueba de caja blanca

Se pretende comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar

Pruebas de código: cubrimiento

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

Considerando que esta función forma parte de un programa mayor, se considera lo siguiente:

- 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 como *prueba(1,1)*, ya que en esta caso, cada línea de la función se ejecuta, incluida z=x;

Pruebas de código: cubrimiento

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

Considerando que esta función forma parte de un programa mayor, se considera lo siguiente:

- Si invocamos a la función con *prueba(1,1)* y *prueba(0,1)*, se satisfará el **cubrimiento de decisión**. En el primer caso, la *if* condición va a ser verdadera, se va a ejecutar z=x, pero en el segundo caso, no.
- El **cubrimiento de condición** puede satisfacerse si probamos con *prueba(1,1)*, *prueba(1,0)* y *prueba(0,0)*. En los dos primeros casos (x<0) se evalúa a verdad mientras que en el tercero, se evalúa a falso. Al mismo tiempo, el primer caso hace (y>0) verdad, mientras el tercero lo hace falso.

Pruebas de código: valores límite

```
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;
}
```

En el código Java adjunto, aparecen dos funciones que reciben el parámetro x

En la *funcion1*, el parámetro es de tipo **real** y en la *funcion2*, el parámetro es de tipo **entero**

Como se aprecia, **el código de las dos funciones es el mismo**, sin embargo, los casos de prueba con **valores límite va a ser diferente**

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

Pruebas de código: valores límite

```
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;
}
```

Por ejemplo, supongamos que queremos probar el resultado de la ejecución de una función, que recibe un parámetro x:

- Si el parámetro x de entrada es **real**, los **valores límite pueden ser 4,99 y 5,01**.
- Si el parámetro de entrada x es un valor **entero**, los **valores límite serán 4 y 5**.

Pruebas de código: valores límite

Si en un bucle while la condición es while (x>5 && x < 10), siendo x un valor real, sería valores límite

- 4 y 11
- 4,99 y 11
- 4,99 y 9,99

Pruebas de código: valores límite

Si en un bucle while la condición es while (x>5 && x < 10), siendo x un valor real, sería valores límite

- 4 y 11
- 4,99 y 11
- **4,99 y 9,99**

Pruebas de código: valores límite

¿Y si el x tiene un valor entero?

Pruebas de código: valores límite

¿Y si el x tiene un valor entero?

Podrían ser valores límite 4, 5, 9 o 10.

Pruebas de código: clases de equivalencia

Tipo de prueba funcional, en donde cada caso de prueba, pretende cubrir el **mayor número de entradas posible**

El **dominio de valores de entrada**, se divide en número finito de clases de equivalencia

Como la entrada está dividida en un c**onjunto de clases de equivalencia**, la prueba de un **valor representativo** de cada clase, permite suponer que el **resultado** que se obtiene con él, será el **mismo** que con cualquier otro valor de la clase

Pruebas de código: clases de equivalencia

Cada clase de equivalencia debe cumplir:

- Si un parámetro de entrada debe estar comprendido entre un **determinado rango**, hay **tres clases de equivalencia**: por debajo, dentro y por encima.
- Si una entrada requiere un **valor entre los de un conjunto**, aparecen **dos clase de equivalencia**: en el conjunto o fuera de él.
- Si una entrada es **booleana**, hay **dos clases**: sí o no.
- Los **mismos criterios se aplican a las salidas esperadas**: hay que intentar generar resultados en todas y cada una de las clases.

Pruebas de código: clases de equivalencia

En este ejemplo, las clases de equivalencia serían:

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

y los respectivos casos de prueba, podrían ser:

1. Por debajo: x=0
2. Dentro: x=50
3. Por encima: x=100

Normas de calidad

Los estándares que se han venido utilizando en la fase de prueba de software son:

Estándares **BSI**:

- BS 7925-1, Pruebas de software. Parte 1. Vocabulario.
- BS 7925-2, Pruebas de software. Parte 2. Pruebas de los componentes software.

Estándares **IEEE** de pruebas de software:

- IEEE estándar 829, Documentación de la prueba de software.
- IEEE estándar 1008, Pruebas de unidad.
- Otros estándares ISO / IEC 12207, 15289.

Normas de calidad

Sin embargo, estos estándares **no cubren determinadas facetas de la fase de pruebas**, como son la organización, el proceso y gestión de las pruebas, presentan pocas pruebas funcionales y no funcionales etc.

Ante esta problemática, la industria ha desarrollado la norma ISO/IEC 29119

Una norma para unificarlas a todas

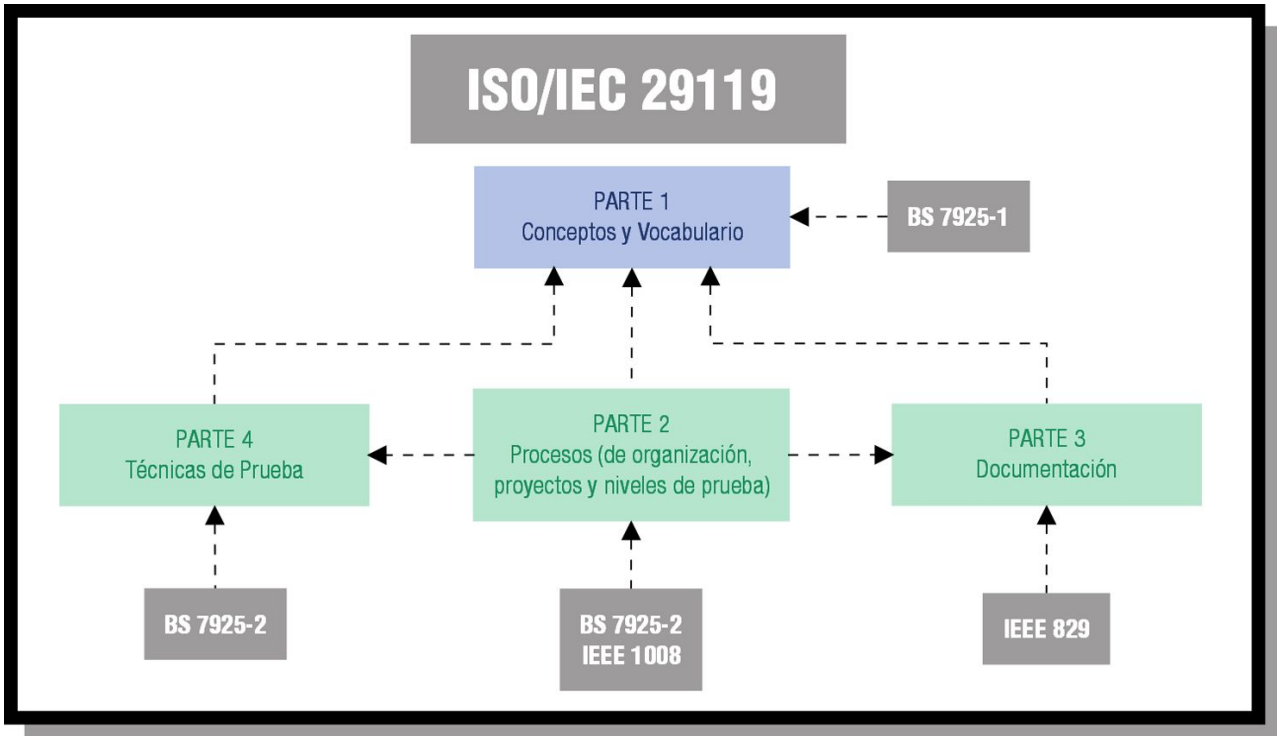
Normas de calidad

La norma **ISO/IEC 29119** de prueba de software, pretende **unificar en una única norma**, todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del software

Desde estrategias de prueba para la organización y políticas de prueba, prueba de proyecto al análisis de casos de prueba, diseño, ejecución e informe

Con este estándar, se podrá realizar cualquier prueba para cualquier proyecto de desarrollo o mantenimiento de software

Normas de calidad



Normas de calidad: ejercicio 1

Realiza una comparativa de los organismos de normalización ISO, IEEE y BSI teniendo en cuenta

- Siglas
- Área de aplicación (nacional, internacional, local…)
- Ámbito que trabaja (seguridad, comunicación, transporte...)
- Productos que estandariza
- Ejemplos de algunas normas

Normas de calidad: ejercicio 2

La norma ISO/IEC 29119

- ¿En qué consiste?
- Finalidad
- Grupo de Trabajo
- Definición de sus cuatro partes
  - Parte 1: Definiciones y Vocabulario.
  - Parte 2: Proceso de Pruebas.
  - Parte 3: Documentación de Pruebas.
  - Parte 4: Técnicas de Pruebas.

Pruebas unitarias

Objetivo de probar el **correcto funcionamiento de un módulo de código**. El fin que se persigue, es que **cada módulo funciona correctamente por separado**.

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

Posteriormente, con la **prueba de integración**, se podrá asegurar el **correcto funcionamiento del sistema**.

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

Pruebas unitarias

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.
- **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.

Pruebas unitarias

Herramientas para Java

- Jtiger
- TestNG
- JUnit

Automatización de la prueba

Los entorno de desarrollo, integran **frameworks**, que permiten automatizar las pruebas.

En el caso de entornos de desarrollo para Java, como NetBeans y Eclipse, nos encontramos con el framework **JUnit**.

Automatización de la prueba

JUnit es una **herramienta de automatización de pruebas** que nos permite de manera rápida y sencilla, elaborar pruebas.

Nos permite **diseñar clases de prueba**, para cada clase diseñada en nuestra aplicación. Una vez creadas las **clases de prueba,** establecemos los **métodos que queremos probar**, y para ello diseñamos **casos de prueba**.

Los criterios de creación de casos de prueba, pueden ser muy diversos, y dependerán de **lo que queramos probar**.

Documentación de la prueba

Como en otras etapas y tareas del desarrollo de aplicaciones, la **documentación** de las pruebas es un requisito indispensable para su correcta realización.

Unas pruebas bien documentadas podrán también servir como base de conocimiento para **futuras tareas de comprobación**.

Las metodologías actuales, como Métrica v.3, proponen que la **documentación de la fase de pruebas** se basen en los estándares ANSI / IEEE sobre verificación y validación de software.

Documentación de la prueba

El propósito de estos estándares es describir un **conjunto de documentos para las pruebas de software**.

Un documento de pruebas **estándar** puede f**acilitar la comunicación entre desarrolladores** al suministrar un marco de referencia común.

La definición de un documento estándar de pruebas puede servir para **comprobar que se ha desarrollado todo el proceso de pruebas de software**.

Documentación de la prueba

Los documentos que se van a generar son:

- **Plan de Pruebas**: Al principio se desarrollará una planificación general. Se inicia el proceso de Análisis del Sistema.
- **Especificación del diseño de pruebas**. Surge de la ampliación y detalle del plan de pruebas.
- **Especificación de un caso de prueba**. Los casos de prueba se concretan a partir de la especificación del diseño de pruebas.
- **Especificación de procedimiento de prueba**. Una vez especificado un caso de prueba, será preciso detallar el modo en que van a ser ejecutados cada uno de los casos de prueba.

Documentación de la prueba

Los documentos que se van a generar son:

- **Registro de pruebas**. Se registrarán los sucesos que tengan lugar durante las pruebas.
- **Informe de incidente de pruebas.** Para cada incidente, defecto detectado, solicitud de mejora, etc.
- **Informe sumario de pruebas**. Resumirá las actividades de prueba vinculadas a uno o más especificaciones de diseño de pruebas.