Pruebas del software.

Entornos de desarrollo

Índice

Introducción

- A veces, los programas son complejos y es difícil hacer que éstos funcionen correctamente.
- Las **pruebas** del software son un conjunto de técnicas utilizadas para verificar que un programa lleva a cabo su tarea correctamente. Desde el punto de vista opuesto, intentan revelar la existencia de errores.

- Cuando detectamos que existe un error, necesitamos localizarlo llevando a cabo técnicas de **depuración** de código, para luego realizar las modificaciones necesarias que eliminen dicho error.

- La realización de pruebas, **no garantiza** que todos los errores vayan a salir a la luz. La media de tiempo empleado en un proyecto para la realización de pruebas en el desarrollo de un programa, ***suele ser la mitad del tiempo del desarrollo*** del proyecto. El tiempo requerido es proporcional a la calidad del código resultante que queramos obtener.

- Existen dos conceptos que tenemos que tener en cuenta a la hora de realizar pruebas:
  - ✓ **Verificación:** Conjunto de actividades encaminadas a asegurar que el software se ajusta a las especificaciones indicadas: *"¿Estamos construyendo el producto correctamente?"*
  - ✓ **Validación:** Conjunto de actividades encaminadas a asegurar que el software se ajusta a los requisitos del cliente: *"¿Estamos construyendo el producto correcto?"*

Introducción

¿Cuántos tests necesitamos?

- Dependiendo del tamaño del programa, por ejemplo, una única clase normalmente podrá ser probado de una sola vez.

- Sin embargo, un programa de mayor tamaño, mayor cantidad de clases, es posible que por su complejidad, deba ser probado por partes.

- En Java, el tamaño natural de cada una de esas partes será la clase y será conveniente probar nuestros programas clase por clase. Esto es lo que se conoce como **pruebas de unidad**. Si estas unidades se tratan de interfaces, hablamos de **pruebas de integración**.

- Por último, cuando se realizan pruebas de funcionamiento del programa completo, las pruebas reciben el nombre de **pruebas de integración** o **pruebas de sistema**.

Introducción

Los principios básicos que guían las pruebas de software según el ISTQB son:

- **Principio 1**: Las pruebas demuestran la presencia de errores, pero no su ausencia.
- **Principio 2**: Las pruebas completas no existen.
- **Principio 3**: Las pruebas se iniciarán lo antes posible en el ciclo de vida del software.
- **Principio 4**: La mayor parte de los errores se suelen concentrar en un número reducido de módulos.
- **Principio 5**: Si las pruebas se repiten una y otra vez, con el tiempo el mismo conjunto de casos de prueba ya no encontrará nuevos errores. Los casos de prueba deben ser examinados y revisados periódicamente.
- **Principio 6**: Las pruebas deben adaptarse a las necesidades específicas del contexto.
- **Principio 7**: Un software puede haber pasado todas las fases de pruebas, pero esto no indica que no pueda tener errores que aún no se han logrado identificar.

Índice

- Introducción.
- **Conceptos clave**.
- Pruebas del software.
- Pruebas unitarias automatizadas
- Desarrollo guiado por pruebas
- JUnit

Especificación

- La **especificación** de un programa, viene dada por los requisitos del cliente, es decir, tras las entrevistas realizadas con este y tras un posterior análisis de la información.
- Ejemplo: “El usuario tendrá que introducir la operación a realizar y los operandos necesarios para realizarla”.

- A partir de la frase anterior, podemos preguntarnos:
  - ✓ ¿Cuáles son las operaciones soportadas por el programa?
  - ✓ ¿Qué tipo de operandos permite el programa? ¿Enteros? ¿Reales?
  - ✓ ¿Se permiten números negativos? ¿Cero?

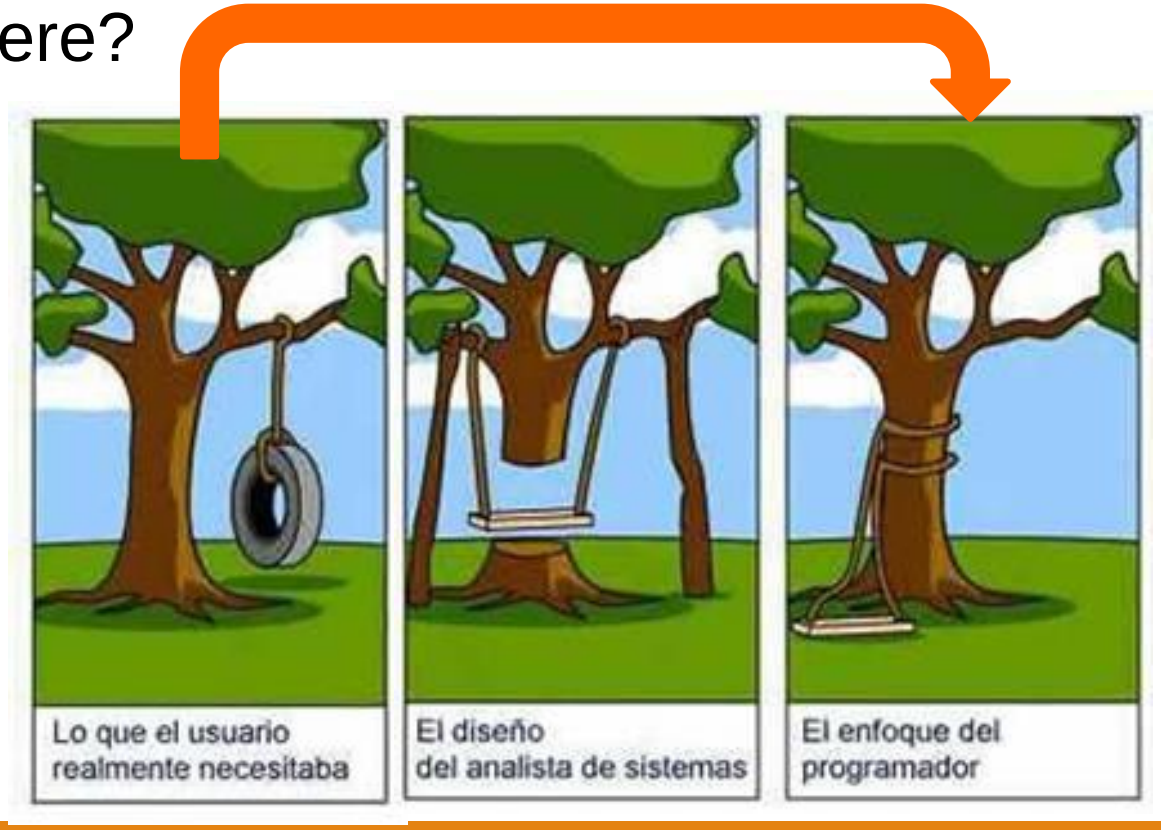
- Muchas veces la información facilitada por el cliente, o por el departamento encargado, no es claro o concisa. Por lo tanto, será parte de nuestro trabajo averiguar y solucionar estos problemas de omisión de información (desconocimiento por parte del cliente, no lo tiene claro, se ha perdido esta información por el camino,…).
- En resumen, una mayor especificación del programa, nos facilitará la tarea de realización de pruebas.

Verificación

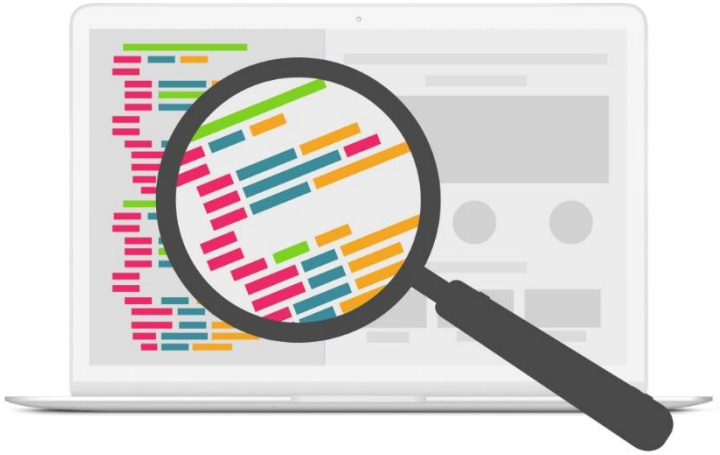
- ¿Estamos construyendo el producto correctamente?
- ¿Estamos siguiendo las especificaciones dadas?

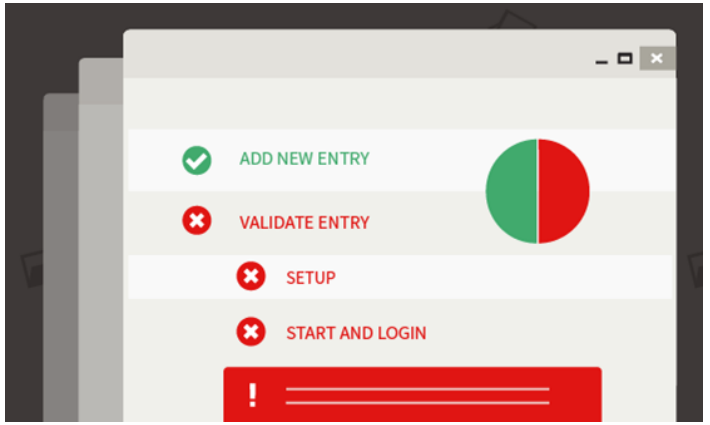
Validación

- ¿Estamos construyendo el producto correcto?
- ¿Estamos siguiendo lo que el cliente quiere?



Métodos



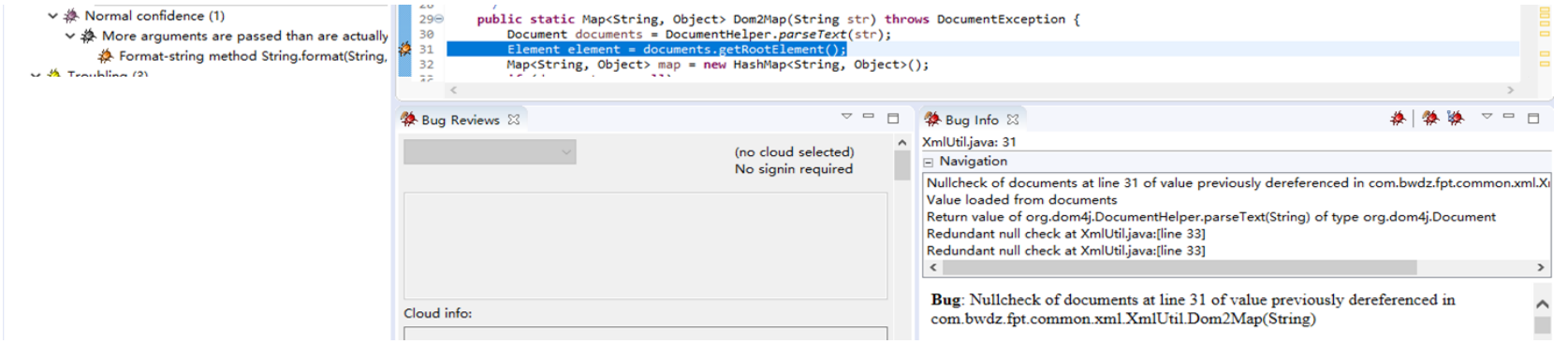


Inspecciones del Software
(Estático)

Pruebas del software
(Dinámico)

Inspecciones del software

- Manualmente
- Automáticamente (ej. FindBugs)
  - ✓FindBugs es una *herramienta de análisis estático* que permite la búsqueda de posibles errores durante el desarrollo de software. Para ello, analiza el bytecode buscando algunos patrones conocidos. No se limita a una búsqueda de expresiones regulares, sino que intenta comprender lo que el programa quiere hacer.
  - ✓FindBugs está disponible para los principales IDEs como IntelllJ, Eclipse …



Índice

- Introducción.
- Conceptos clave.
- **Pruebas del software**.
- Pruebas unitarias automatizadas
- Desarrollo guiado por pruebas
- JUnit

Tipos de Prueba

Caso de prueba:

Dados unos valores de entrada, tenemos que obtener unos resultado tipo, es decir, tiene que validar unas condiciones de prueba.

Técnica de diseño

- Caja Negra
- Caja Blanca

Automatización

- Manuales
- Automáticas

Fases prueba

- Unitarias
- Integración
- Sistema
- Aceptación
- Regresión

Técnicas de diseño

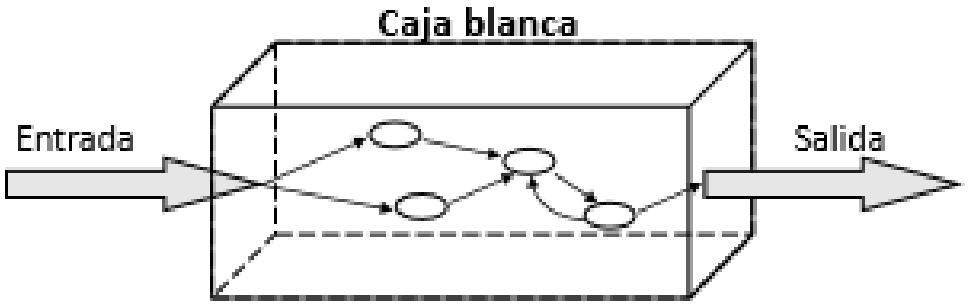
- Llegados a este momento, tenemos que discernir entre dos conceptos diferentes:
  - ✓ **Prueba**: Es el proceso de ejecución de un programa con el intento deliberado de encontrar errores.
  - ✓ **Caso de prueba**: Conjunto de entradas, condiciones de ejecución y resultados esperados diseñados para un objetivo particular de condición de prueba.

- Existen dos técnicas de diseño por referencia:
  - ✓ Caja blanca
  - ✓ Caja negra

Técnicas de diseño: Caja blanca

Caja blanca o pruebas estructurales:

- Este otro tipo de pruebas se basa en el **conocimiento del funcionamiento interno** del programa, la estructura del mismo para seleccionar los datos de prueba.
- Es decir, tenemos en cuenta cada declaración, cada bifurcación de una estructura de selección, cada iteración de cada estructura de control, …
- Este tipo de pruebas, se basan en unos criterios de cobertura lógica, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores.



Técnicas de diseño: Caja blanca

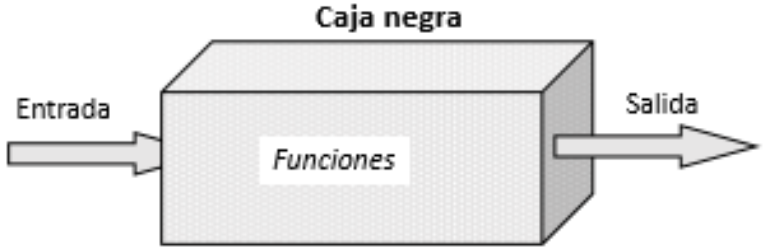
Las técnicas más usuales de este tipo de pruebas son:

- **Prueba de interfaz**: Este tipo de prueba debe ser la primera en realizar. Se basa en analizar el flujo de datos que pasa a través de la interfaz del módulo, tanto externa como interna, para asegurar que la información fluye de forma adecuada tanto hacia el interior como hacia el exterior del módulo que se está probando.
- **Prueba del camino básico**: Permite obtener una medida de la complejidad lógica de un programa (**complejidad ciclomática**) y utilizar esa medida como guía para definir un conjunto básico de caminos de ejecución. Es decir, este tipo de prueba está orientado a cubrir la ejecución de cada una de las sentencias, cada una de las decisiones y cada una de las condiciones en las decisiones, tanto en su vertiente verdadero como falsa. Es decir, tenemos en cuenta cada declaración, cada bifurcación de una estructura de selección, cada iteración de cada estructura de control, …
- **Prueba de bucles**: Comprueban la validez de las construcciones de los bucles. Determinar la validez de las construcciones de los bucles es fundamental para garantizar que es correcto el módulo que se está probando.

Técnicas de diseño: Caja negra

Caja negra o pruebas funcionales:

- Este tipo de pruebas se basa en ***utilizar unos datos de entrada que pueden ser representativos de todos los posibles datos de entrada***. En ningún momento, tenemos conocimiento del funcionamiento interno del programa. Solamente somos conscientes de las entradas y de las salidas.
- *Solamente somos conscientes de la funcionalidad que debe ofrecer el programa independientemente de su funcionamiento interno*. Es decir, si hay excepciones en los datos a procesar, hay que procurar que estas sean procesadas, así como los datos de entrada correspondientes a los límites máximos y mínimos especificados.



Técnicas de diseño: Caja negra

Las técnicas más usuales de este tipo de pruebas son:

- **Particiones de equivalencia**: Consiste en dividir el dominio de entrada de un programa en clases de equivalencia de los que se pueden derivar casos de prueba, donde la prueba de un valor representativo de la misma sea extrapolable al que se conseguiría probando cualquier valor de la clase.
- **Análisis de valores límites**: A la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia. Es más fácil que existan errores en los límites que en el centro. Ejemplo: un campo numérico que requiera una cifra de 3 dígitos, tendrá como valores límites inferiores 99 y 100, y como valores límites superiores 999 y 1000.
- **Valores típicos de error**: Desarrolla casos de prueba con ciertos valores susceptibles de causar problemas, esto es, valores típicos de error, y valores especificados como no posibles. Se realiza en función de la naturaleza y funcionalidad del programa a probar, por lo que depende en buena medida de la experiencia del diseñador de la prueba.

Fases del proceso de prueba

- **Prueba unitaria o pruebas de unidad:**
  - ✓ Son las pruebas iniciales del sistema y las demás pruebas deben apoyarse en ellas.
  - ✓ Tienen por objetivo verificar la funcionalidad y estructura de cada componente (**módulo**) individualmente, una vez que ha sido codificado. Es decir, pretenden comprobar que el módulo, entendido este como una unidad funcional de un programa completamente independiente, está correctamente codificado.

- **Pruebas integrales** o **pruebas de integración**:
  - ✓ Estas pruebas son necesarias ya que las pruebas unitarias no aseguran que cuando se integren todos los módulos como un conjunto, el sistema completo funcione. La forma de ensamblar los módulos introduce una complejidad adicional a estas pruebas, por lo que, para paliar en la medida de lo posible, los problemas de integración se han definido las siguientes estrategias:

    a) **Integración incremental**: Los módulos se integran poco a poco y a medida que se integran se van probando. Caben dos posibilidades:

    Integración incremental descendente (top-down)

    Integración incremental ascendente (bottom-up)

    b) **Integración no incremental**: los módulos se integran de una vez y se prueban todos al mismo tiempo.

- **Pruebas del Sistema:**
  - ✓ Una vez validado el software de forma aislada mediante las pruebas unitarias y pruebas de integración, hay que combinar el software con el resto de componentes del sistema (hardware, base de datos, etc.)
  - ✓ Las pruebas del sistema verifican que cada elemento encaja de forma adecuada y que se alcanza la funcionalidad y el rendimiento del sistema completo. De esta forma se obtiene una visión global del sistema similar a la que se mostraría en un entorno real de producción.

Fases del proceso de prueba

- **Pruebas de aceptación:**
  - ✓ Son aquellas en las que realiza el cliente para comprobar si el sistema cumple los requisitos expresados por él, de modo que el sistema esté listo para el uso operativo en el entorno en el que se va a explotar el sistema.
  - ✓ Podemos distinguir:
    - **Pruebas alfa**: los clientes prueban el programa en el lugar de desarrollo con la presencia del equipo desarrollador, mientras éste registra errores. Es decir, las pruebas alfa se llevan a cabo en un entorno controlado.
    - **Pruebas beta**: los clientes prueban el programa en su lugar de trabajo y sin la presencia del equipo de desarrollo. El cliente registra errores durante la prueba y va informando al equipo de desarrollo.

- **Pruebas de Regresión**:
  - ✓ Las pruebas de regresión están asociadas a la fase de mantenimiento del software y su objetivo es comprobar que los cambios efectuados no producen un comportamiento no deseado en el programa.
  - ✓ Las pruebas de regresión se deben realizar cada vez que se efectúa un cambio en el sistema, ya sea para corregir un error o para introducir una mejora, a fin de controlar que las modificaciones no producen efectos negativos sobre el mismo u otros componentes. Normalmente implican la repetición de las pruebas que ya se han realizado previamente (unitarias, de integración, de aceptación, etc.).

Pruebas del software: Automatización

- Algunas pruebas de software tales como las pruebas de regresión intensivas de bajo nivel pueden ser laboriosas y consumir mucho tiempo para su ejecución si se realizan manualmente.
- Adicionalmente, una aproximación manual puede no ser efectiva para encontrar ciertos tipos de defectos, mientras que las pruebas automatizadas ofrecen una alternativa que lo permite.
- Una vez que una prueba ha sido automatizada, esta puede ejecutarse repetitiva y rápidamente en particular con productos de software que tienen ciclos de mantenimiento largo, ya que incluso cambios relativamente menores en la vida de una aplicación pueden inducir fallos en funcionalidades que anteriormente operaban de manera correcta. Existen dos aproximaciones a las pruebas automatizadas

Pruebas del software: Automatización

Existen dos aproximaciones a las pruebas automatizadas:

- **Pruebas manejadas por el código**: Se prueban las clases, módulos o bibliotecas con una variedad amplia de argumentos de entrada y se valida que los resultados obtenidos sean los esperados.
- **Pruebas de Interfaz de Usuario**: Un marco de pruebas genera un conjunto de eventos de la interfaz de usuario, tales como teclear, hacer click con el ratón e interactuar de otras formas con el software y se observan los cambios resultantes en la interfaz de usuario, validando que el comportamiento observable del programa sea el correcto.

Índice

- Introducción.
- Conceptos clave.
- Pruebas del software.
- **Pruebas unitarias automatizadas.**
- Desarrollo guiado por pruebas.
- Junit.

Pruebas unitarias automatizadas.

¿Qué probamos? - Según el tipo de componente

- Funciones individuales o métodos estáticos
  - ✓ Clases de objetos
  - ✓ Pruebas aisladas de los métodos
  - ✓ Asignación y consulta de atributos
  - ✓ Pruebas de secuencias de operaciones
- No olvidar probar también el manejo de errores (**Excepciones**).

Pruebas unitarias automatizadas.

Funciones individuales

- ✓Probar entradas / salidas

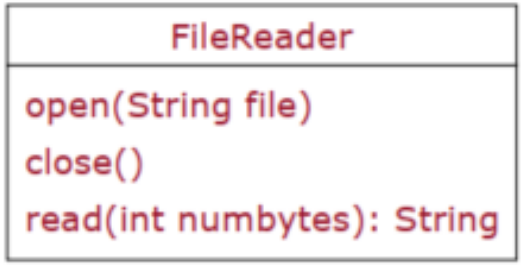
```
@Test
Public void factorial1() {
        assert factorial(0) == 1;
}

@Test
Public void factorial2() {
        assert factorial(1) == 1;
}

@Test
Public void factorial3() {
        assert factorial(4) == 24;
}
```

Pruebas unitarias automatizadas.

- Pruebas aisladas de las operaciones
- Secuencias de operaciones
  - ✓ Open – Read – Close
  - ✓ Close – Read
  - ✓ Open – Close – Read
  - ✓ …




```
@Test
Public void filereader1() {
   file.open("isg2");
   String s = file.read(10);
   file.close();
   assert s.equals("Hola mundo");
}
@Test(expected=FileNotOpenException.class)
Public void filereader2() {
   file.read(10);
   file.close();
}
@Test(expected=FileNotOpenException.class)
Public void filereader3() {
   file.open("isg2");
   file.close();
   String s = file.read(10);
}
```

Pruebas unitarias automatizadas.

¿Qué probamos?

- **Casos de prueba positivos:** Probar que las cosas funcionan bien. Todo test positivo debe tener un *assert* para comprobar que el test es pasado:

```
@Test
Public void filereader1() {
   file.open("isg2");
   String s = file.read(10);
   file.close();
   assert s.equals("Hola mundo");
}
```

- **Casos de prueba negativos:** Probar que el sistema maneja bien los errores.

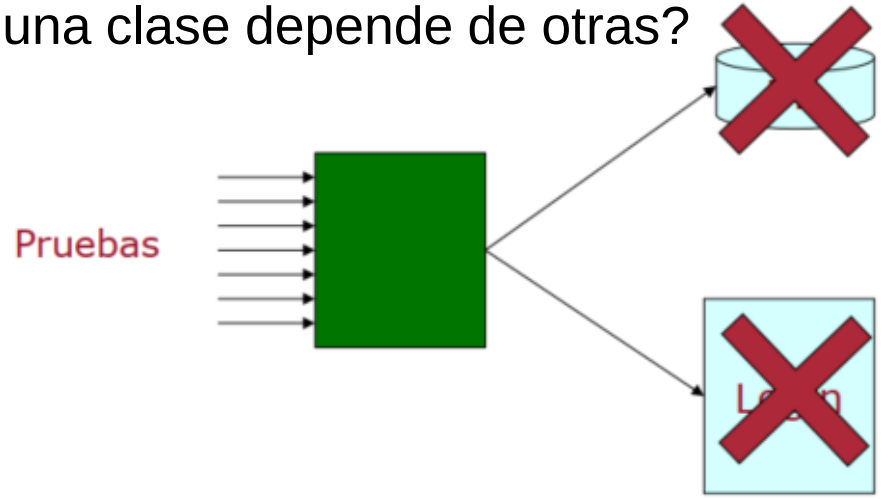
```
@Test(expected=FileNotOpenException.class)
Public void filereader2() {
   file.read(10);
   file.close();
}
```

Pruebas unitarias automatizadas.

¿Qué probamos?

- En las pruebas unitarias debemos considerar la clase de forma aislada, sin importarnos el resto del sistema:

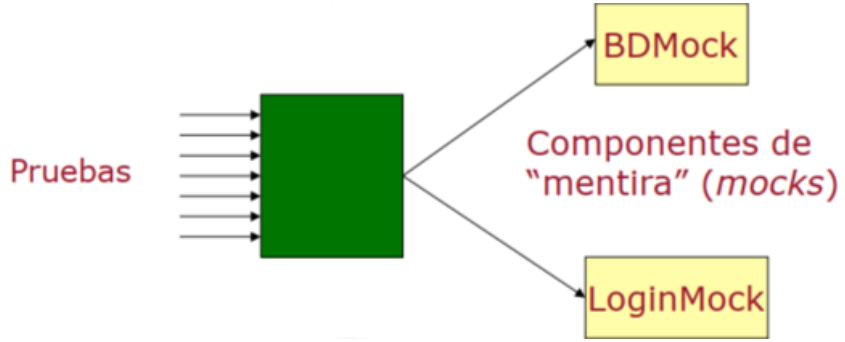
- ¿Qué pasa cuando una clase depende de otras?



Pruebas unitarias automatizadas.

¿Qué probamos?

Es conveniente sustituirlas por objetos *mock*, si no, no estamos haciendo pruebas unitarias propiamente dichas:




```
public class LoginMock implements ILogin {
   boolean login(String name, String pass){
      if (name.equals("isg2")) return true;
         else return false;
   }
}
```

Los **Mock** son objetos que imitan el comportamiento de objetos reales de una forma controlada. Se usan para probar a otros objetos en pruebas unitarias que esperan mensajes de una clase en particular para sus métodos, al igual que los diseñadores de autos usan un *crash dummy* cuando simulan un accidente.

Pruebas unitarias automatizadas.

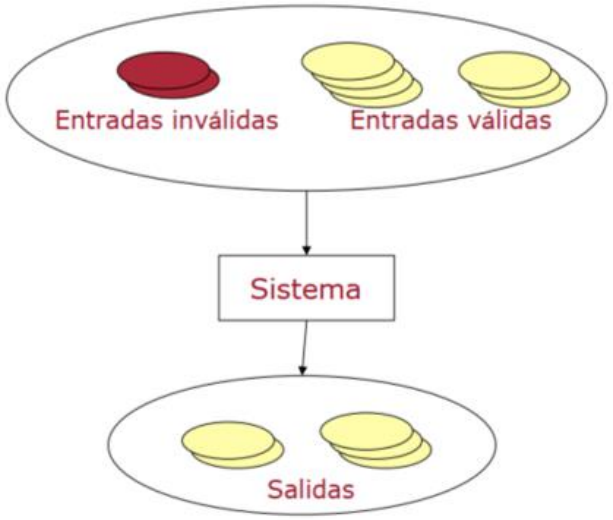
¿Cómo hacemos las pruebas unitarias?

- Existen muchas alternativas
  - ✓ Hacer buenas pruebas depende de la intuición y la experiencia
- Ideas a la hora de hacer pruebas:
  - ✓ Pruebas de particiones
  - ✓ Pruebas estructurales

Pruebas unitarias automatizadas.

Pruebas de particiones

```
public interface ILogin {
   boolean login(String name, String pass);
}
```



- Usuarios que no existen (resultado falso)
- Usuarios que existen (resultado cierto)
- Entradas inválidas: ¿Qué pasa si *name* es vacío?

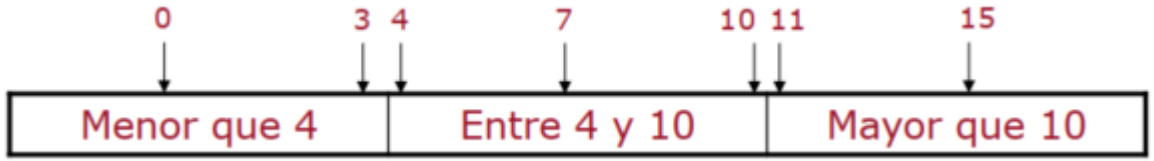
```
@Test
public void loginvalido() {
        assert login("prueba","isg2") == true;
}
@Test
public void logininvalido() {
        assert login("prueba","prueba") == false;
}
```

Pruebas unitarias automatizadas.

Pruebas de particiones

Probar siempre los límites:

- Ejemplo: Supongamos que tenemos una función devuelve resultados distintos dependiendo de si la entrada es menor que 4, entre 4 y 7 y mayor que 7.
  ¿Qué valores probamos como entrada?

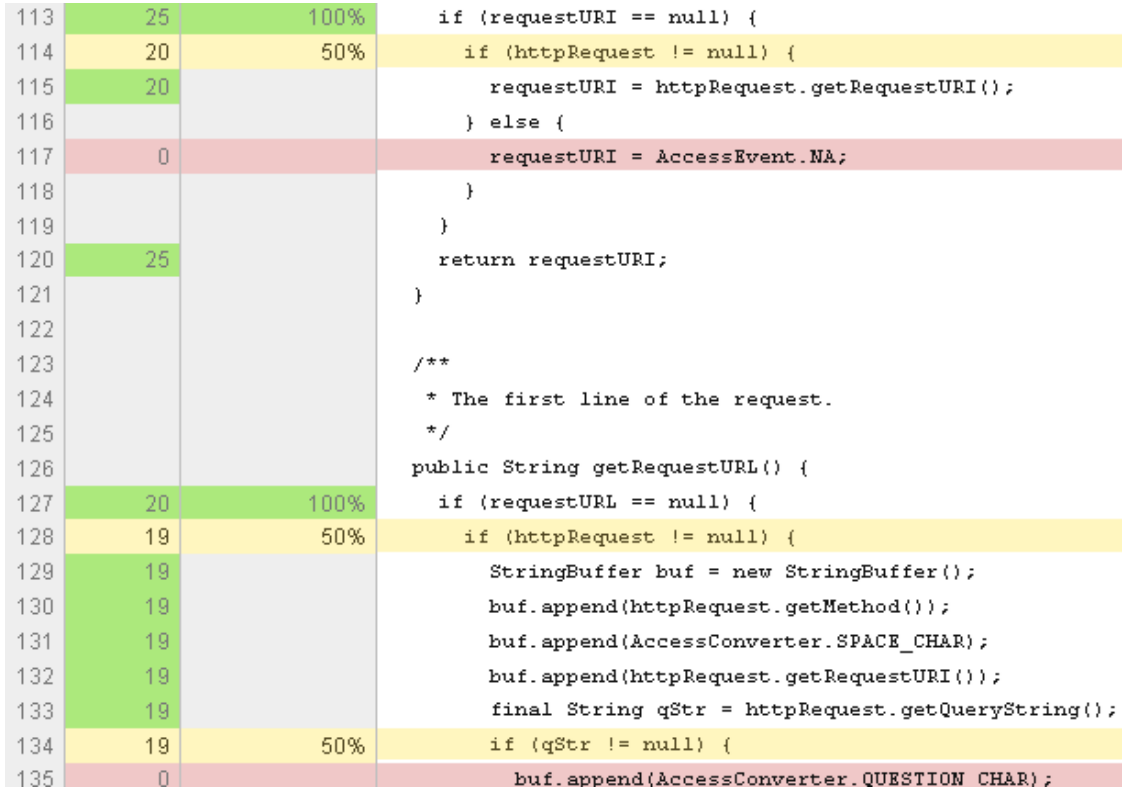


- Cuando tenemos listas, vectores, tablas:
  - ✓ Listas de un solo valor y vacías.
  - ✓ Probar siempre distintos tamaños.
  - ✓ Comprobar primer elemento, elemento central y último elemento.
  - ✓ Pasar *null* en vez del objeto.

Pruebas unitarias automatizadas.

Pruebas estructurales

- Son de caja blanca
- Preparo las pruebas para intentar ejecutar cada sentencia al menos una vez .



113	25	100%	if (requestURI == null) {
114	20	50%	if (httpRequest != null) {
115	20		requestURI = httpRequest.getRequestURI();
116			} else {
117	0		requestURI = AccessEvent.NA;
118			}
119			}
120	25		return requestURI;
121			}
122			
123			/**
124			* The first line of the request.
125			*/
126			public String getRequestURL() {
127	20	100%	if (requestURL == null) {
128	19	50%	if (httpRequest != null) {
129	19		StringBuffer buf = new StringBuffer();
130	19		buf.append(httpRequest.getMethod());
131	19		buf.append(AccessConverter.SPACE_CHAR);
132	19		buf.append(httpRequest.getRequestURI());
133	19		final String qStr = httpRequest.getQueryString();
134	19	50%	if (qStr != null) {
135	0		buf.append(AccessConverter.QUESTION_CHAR);

¿Dónde ejecutamos las pruebas unitarias?

- Software para realizar pruebas automáticas (para Java):
  - ✓JUnit
  - ✓TestNG
- Se integran con entornos de desarrollo como Eclipse
  - ✓Hay frameworks para ayudar a la creación de objetos *mock:*
  - ✓Moq, jMock, EasyMock, Typemock, jMockit, Mockito o PowerMock

¿Cuándo implementamos las pruebas unitarias?

- De forma paralela a la codificación.
- Incluso antes siguiendo desarrollo guiado por pruebas.

Índice

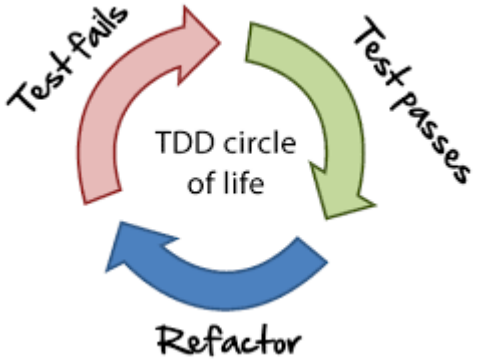
- Introducción.
- Conceptos clave.
- Pruebas del software.
- Pruebas unitarias automatizadas.
- **Desarrollo guiado por pruebas.**
- JUnit

Desarrollo guiado por pruebas

Test Driven Development (**TDD**) es una metodología de desarrollo basada en pruebas. A grandes rasgos, esta metodología comprende dos fases:

- **Test First Development**: Las pruebas se elaboran antes de la fase de codificación, pruebas unitarias. La batería de pruebas es verificada antes de pasar a la siguiente fase.
- **Refactoring** (Refactorización): Proceso de reestructuración del código, código interno, sin que afecte a los resultados (comportamiento externo). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés).

La finalidad de este modelo de desarrollo es lograr un código limpio que funcione. Al traducir los requisitos a pruebas, se garantiza que el software cumple con los requisitos establecidos.



Desarrollo guiado por pruebas

Desarrollo Habitual

1. Elijo una funcionalidad
2. Programo el código
3. Implemento la prueba
4. Ejecuto la prueba
5. Corrijo los errores

Desarrollo guiado por pruebas:

1. Elijo una funcionalidad
2. Implemento la prueba
3. Ejecuto los tests para ver si el nuevo falla
4. Programo el código
5. Ejecuto los tests
6. Refactorizo

Desarrollo guiado por pruebas

Con respecto el paso 4 (Programar el código):

- ✓ El tamaño del paso debe ser pequeño. Es decir, programar poco código entre prueba y prueba.
- ✓ Sólo se debe escribir el código necesario para pasar la nueva prueba. Ningún otro código adicional.
- ✓ El código nuevo puede no ser muy elegante o eficiente (mal diseño). Eso se resuelve en la refactorización.
- ✓ Refactorizar es MUY IMPORTANTE.

Ventajas del TDD

- ✓Comprensión.
- ✓Documentación.
- ✓Evita sobreingeniería.

Índice

- Introducción.
- Conceptos clave.
- Pruebas del software.
- Pruebas unitarias automatizadas.
- Desarrollo guiado por pruebas.
- **JUnit**

JUnit

JUnit es un conjunto de clases (**framework**) que se utilizan a la hora de programar para realizar las pruebas unitarias de aplicaciones Java.

Realiza ejecuciones de las clases para comprobar que el funcionamiento de éstas es totalmente correcto, en caso contrario, se notificará un fallo en el método correspondiente.

El uso de esta herramienta se ha extendido por:

- ✓Incrementan la calidad y velocidad de generación de código.
- ✓La escritura de test es sencilla.
- ✓Los tests chequean sus propios resultados y proporcionan información de retorno al instante.
- ✓Los tests incrementan la estabilidad del software.
- ✓Estos se escriben en Java.
- ✓Y es una herramienta Gratuita.

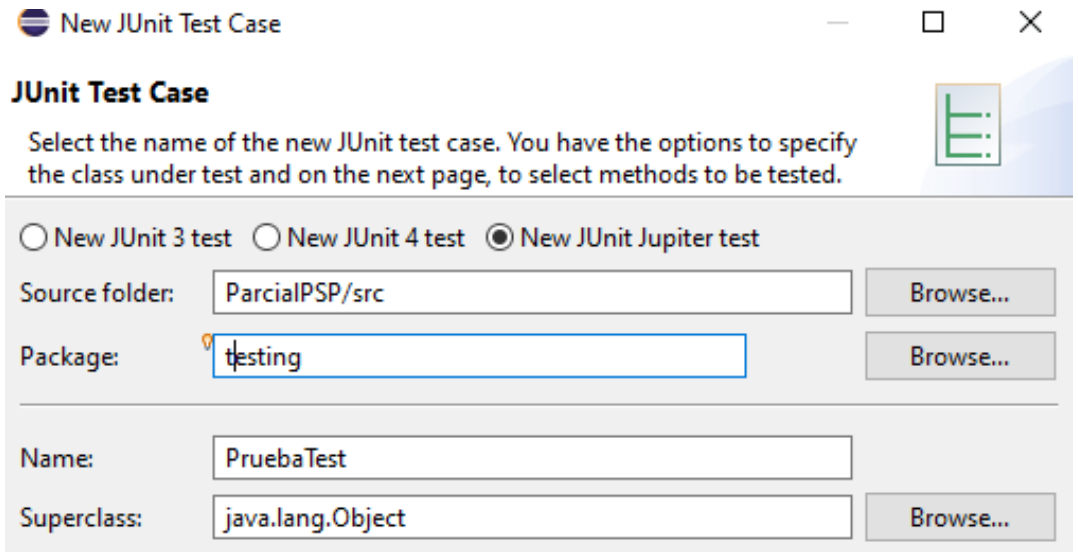
Integración de JUnit en Eclipse

Antes de empezar a trabajar con JUnit, vamos a dejar claro que las contenidos venideros están realizados con la versión 5.

Para ello:

1) Nos dirigimos a nuestro proyecto, hacemos click en el botón derecho de nuestro paquete y seleccionamos JUnit Test Case.

2) Seleccionamos:



Anotaciones en JUnit

JUnit se basa en anotaciones para marcar aquellos métodos de prueba

Anotación	Descripción
@Test	Indica a JUnit que se trata de un método de test.
@BeforeEach	Se ejecuta siempre antes de cada método de test. Se suele utilizar para preparar el entorno de prueba (por ejemplo: leer datos de entrada, inicializar la clase, etc..)
@AfterEach	Se ejecuta siempre después de cada método de test. Se suele utilizar para limpiar el entorno de desarrollo (por ejemplo: borrar datos temporales, restaurar valores por defecto, etc..)
@BeforeAll	Se ejecuta solo una vez, antes de la ejecución de todos los tests. Se suele utilizar para ejecutar actividades de inicio muy costosas (por ejemplo: conexión a una base de datos). Estos métodos se tienen que definir como estáticos.
@AfterAll	Se ejecuta solo una vez, después de la ejecución de todos los tests. Se suele utilizar para ejecutar actividades de cierre (por ejemplo: desconexión a una base de datos). Estos métodos se tienen que definir como estáticos.
@Disable	Indica a JUnit que el método de test está deshabilitado. Útil cuando el código ha cambiado sustancialmente y el test no está adaptado. Es de buena práctica indicar el motivo de por qué se ha deshabilitado.

Aserciones en JUnit

Tras crear las condiciones que necesitamos validar, necesitamos comparar si el resultado es el esperado o no. Para este cometido, JUnit dispone de una lista de funciones incluidas en su clase Assert (métodos assert comparan el valor obtenido con el valor esperado, lanzando una excepción si no son iguales).

Método	Descripción
assertTrue(condición booleana,[mensaje])	Comprueba que la condición sea verdadera.
assertFalse(condición booleana,[mensaje])	Comprueba que la condición sea falsa.
assertEquals(valor esperado, valor actual, [mensaje])	Comprueba que dos valores sean iguales. Nota: en arrays comprueba su referencia, no el contenido!)
assertSame(valor esperado, valor actual, [mensaje])	Comprueba que ambos parámetros sean el mismo objeto.
assertNotSame(valor esperado, valor actual, [mensaje])	Comprueba que ambos parámetros no sean el mismo objeto.
assertNull(objeto, [mensaje],)	Comprueba que el objeto sea nulo.
assertNotNull(objeto, [mensaje])	Comprueba que el objeto no sea nulo.
fail([mensaje])	Hace que el método falle. Debería ser utilizado solo para comprobar que una parte del código de test no se ejecute o para hacer fallar un test no implementado.