"Todos somos muy ignorantes, lo que ocurre es que no todos ignoramos las mismas cosas."
Albert Einstein

INTRODUCCIÓN A LOS TESTS UNITARIOS

El testing es una práctica que tiene que ver con la calidad del código que estamos construyendo. Hemos visto muchas buenas prácticas que inciden en la calidad de código. Sólo por destacar alguna de ellas podemos citar: divide y vencerás (funciones con una alta cohesión y bajo acoplamiento), nombrado de identificadores, indexado correcto del código, documentación interna y externa, etc.

En el desarrollo software es fácil encontrar que una aplicación de tamaño medio podría tener del orden de cinco mil clases, cien mil métodos y unas cinco millones de líneas de código. De todo el tiempo de desarrollo, se estima que aproximadamente el 50% se emplea para la realización de tests o pruebas y una porción muy importante de ese esfuerzo de testing se destina a la realización de tests unitarios.

La etapa de pruebas de software (software testing) es un proceso más dentro del ciclo de vida clásico de desarrollo. Sirve para mantener un control de calidad sobre el producto que creamos y, aunque requiera una gran cantidad de tiempo y esfuerzo, a la larga **el software mal probado acaba resultando demasiado caro**.

Caso de Pruebas

Un caso de prueba (test case) es una situación, contexto o escenario bajo el que se comprueba una funcionalidad de un programa para ver si se comporta de la forma en qué se espera.

Pongamos como ejemplo un videojuego en el que cuando el personaje cae al agua, éste pierde una vida y el nivel se reinicia.

- **Objeto de la prueba**: asegurarnos que cuando el personaje cae al agua, ocurre lo esperado
- **Caso prueba**: llevas al personaje a un punto de la pantalla donde pueda caer al agua

Otro ejemplo lo podríamos tener en la API de Java, int Math.abs(int num) devuelve un int con el valor absoluto del parámetro num. Si num > 0, devuelve num, si num < 0, devuelve -num.

- **Objeto de la prueba**: asegurarnos que Math.abs() devuelve el valor absoluto
- **Caso de prueba**: si calculo el valor absoluto de -7 debe dar –(-7), es decir 7.

En otras palabras, un caso de prueba es una pregunta que se le hace al programa o función para saber si el programa o función contesta (reacciona) correctamente (tal y como se quiere que reaccione).

Tipos de pruebas según su enfoque

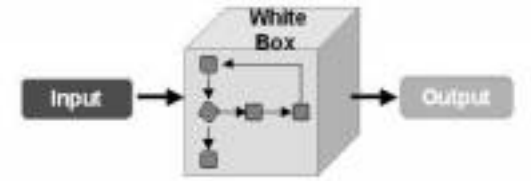
Caja Negra

Son las pruebas que se centran en evaluar el valor de las salidas de un sistema a partir de unas entradas concretas, sin tener en cuenta el funcionamiento interno del sistema. Se centran en el **qué** hace un sistema y no en el **cómo** lo hace. **El desarrollador de la prueba no necesita saber cómo funciona el código, sino evaluar solamente la salida**.

Este tipo de test se puede aplicar a cualquier nivel de testeo de software: *pruebas unitarias*, de *integración*, de *aceptación*, etc.

Caja Blanca

Estos tipos de pruebas se centran en analizar cada uno de los posibles caminos en el flujo de ejecución de un programa o función para unos valores de entrada concretos. **Es decir, si ante unos valores de entrada (parámetros) de un método, el flujo del programa ejecuta los *if*, o los *else*, o entra en un *bucle*, o sale de él.** Este tipo de funcionamiento se puede comprobar también con el depurador, como ya hemos visto en clase.



Las pruebas de Caja Blanca también se pueden aplicar a las *pruebas unitarias*, *pruebas de integración* o más ampliamente a las *pruebas de sistema*.

Tipos de pruebas según su alcance

Pruebas Unitarias

Una prueba unitaria es un tipo de prueba enfocada en verificar una sección específica dentro del código de un programa. En programación orientada a objetos las pruebas unitarias se refieren a las pruebas realizadas sobre los métodos públicos, estáticos o no, de cada una de las clases del sistema. Nosotros nos centraremos en los métodos no estáticos. Dado un método público de una clase, este puede tener más de una prueba asociada.

Las pruebas unitarias es código escrito el mismo lenguaje de programación en el que se desarrolla la aplicación. Debido a la granularidad de las pruebas unitarias, la realización de éstas no asegura el correcto funcionamiento de la aplicación; por el contrario, lo único que podemos asegurar es el correcto funcionamiento del método probado.

Las pruebas unitarias son código (Java, C#, etc) escrito normalmente al mismo tiempo que se va desarrollando el programa (enfoque Caja Blanca). Un método creado en una clase puede tener muchas pruebas unitarias asociadas, como veremos. Las pruebas unitarias por si mismas no aseguran el funcionamiento completo del programa, sin embargo, nos indican que los bloques de software que vamos creando funcionan correctamente, eso sí, independientemente de los otros.

Desde el punto de vista de los desarrolladores, los tests unitarios son las pruebas más importantes para el desarrollo de software.

Para realizar pruebas unitarias tenemos herramientas similares para cada lenguaje o plataforma:

- **JUnit** para **Java**
- **PHPUnit** para **PHP**
- **CPPUnit** para **C++**
- **NUnit** para **.NET**
- **CUnit** para **C**
- **PyUnit** para **Python**

Independientemente de la plataforma, las pruebas unitarias deben cumplir con las siguientes características (Principio **FIRST**):

FAST: Rápida ejecución.

ISOLATED: Independientes de otros tests.

REPEATABLE: Su ejecución se puede repetir en el tiempo tantas veces como se desee.

SELF-VALIDATING: Cada test debe poder validar a sí mismo si es correcto o no.

TIMELY: Para que los tests sean útiles deben estar disponibles en el momento adecuado.

Pruebas de Integración

Las pruebas de integración es una fase en si misma dentro del proceso de pruebas de software, en la que se combinan los distintos módulos de software de un programa y se comprueba que trabajan correctamente de forma conjunta. Las pruebas de integración se llevan a cabo después de la fase de pruebas unitarias y se centran en probar la comunicación entre los componentes, ya sea hardware o software.

Los tests de integración prueban que el sistema completo funciona como se espera. Por ejemplo, una interfaz gráfica y la aplicación con la que trabaja, una aplicación de acceso a datos y el módulo que trabaja un gestor de bases de datos, etc.

Pruebas Funcionales

Son pruebas de **caja negra** sobre todos aquellos componentes, ya sean software o hardware, que tienen que ver con una funcionalidad completa dentro de una aplicación. Desde el punto de vista del usuario final de la aplicación, una **funcionalidad** es un subsistema de la aplicación que cumple con un requisito específico del usuario. Por ejemplo, en una aplicación de comercio electrónico podríamos tener la funcionalidad de log-in en el sitio web. Una funcionalidad o subsistema más de *grano fino*, podríamos considerar el login a través de las credenciales de *Facebook*. En este caso las pruebas funcionales se basan en probar el subsistema formado por todos aquellos componentes software o hardware que intervienen en la funcionalidad de login mediante las credenciales de la red social *Facebook*.

JUnit

Es un framework o marco de trabajo Java enfocado en la realización de pruebas unitarias (unit testing). Consiste en unas librerías (JAR) que debemos añadir a un proyecto en Java. http://www.junit.org. JUnit se encuentra totalmente integrado con las últimas versiones de Eclipse, por lo que no es necesario que realicemos ninguna acción extra para integrar el framework de testing en el IDE Eclipse.

Los tests JUnit se especifican y se gestionan a través de *anotaciones*. Las anotaciones Java se pueden ver como palabras reservadas que dotan de funcionalidad extra a una clase, un método o un atributo de la clase. Estas nuevas palabras reservadas son dependientes del framework, en este caso dependientes de JUnit. Las anotaciones van precedidas del símbolo **@**. El framework JUnit cuenta con multitud de anotaciones que pueden aportar funcionalidad muy variada y útil a nuestros tests, sin embargo, nosotros nos limitaremos a usar las siguientes:

@Test	Los métodos anotados mediante esta anotación son identificados como un caso de test por el framework JUnit.
@Before (JUnit 4), @BeforeEach (JUnit 5)	Los métodos anotados con **@BeforeEach** se ejecutan antes de cada método de test, es decir, se ejecutan antes que los métodos anotados con **@Test**.

@After (JUnit 4), AfterEach (JUnit 5)	Los métodos anotados con **@After** se ejecutan después de cada método de test, es decir, se ejecutan después que los métodos anotados con **@Test**.
@BeforeClass (JUnit 4), @BeforeAll (JUnit 5)	El método anotado con **@BeforeAll** se ejecuta antes (y sólo una vez) de la ejecución de cualquier test. El método anotado con **@BeforeAll** debe ser **static**.
@AfterClass (JUnit 4), @BeforeAll (JUnit 5)	El método anotado con **@AfterAll** se ejecuta después (y sólo una vez) de la ejecución de cualquier test. El método anotado con **@AfterAll** debe ser **static**.
@Ignore (JUnit 4), @Disabled (JUnit 5)	La clase o métodos de una clase anotados con **@Disabled** no serán ejecutados por el framework JUnit.

Ejemplo de ciclo de vida de una clase de test: ***CicloDeVidaTest.java***

Para las validaciones o condiciones de aceptación de los tests emplearemos los siguientes asserts:

assertTrue / assertFalse(condición a testear)	Comprueba que la *condición a testear* sea verdadera o falsa.
assertEquals / asserNotEquals(valor esperado, valor obtenido)	Comprueba la igualdad o no de los 2 valores pasados por parámetro.
assertNull / assertNotNull(objeto)	Comprueba que el objeto pasado por parámetro sea nulo o no.
fail()	Fuerza a que la prueba falle. Se puede especificar un mensaje de fallo.

Simular dependencias en tests unitarios

En ocasiones es necesario simular las dependencias que requiere un método para que éste lleve a cabo su trabajo. Es decir, dado un *metodoA* de la *ClaseA*, si el *métodoA* requiere o invoca a otro *metodoB* de la *ClaseB*, cuando se realicen las pruebas del *metodoA*, se deben simular las invocaciones que desde el métodoA se realicen al método. En otro caso, estas pruebas sobre el *metodoA* dependerán también de los resultados de la invocación el *metodoB*. Con lo que: **NO SERÍA UN TEST UNITARIO SINO UNA PRUEDE INTEGRACIÓN!!!**

Para ello, todas aquellas clases que se deben simular deberán ser anotadas con la anotación **@Mock**. Adicionalmente, dichos *mocks* o clases simuladas deberán ser *inyectadas* en la clase que está siendo objeto de test por medio del uso de la anotación **@InjectMocks.** El último paso que se debe realizar es inicializar las anotaciones anteriores en el cuerpo de un método anotado como **@BeforeEach. (**Ejemplo en ***EcuacionPrimerGradoTest.java***)