JUnit

Las pruebas constituyen una fase costosa y laboriosa dentro del proceso de desarrollo del software y cada vez las empresas hacen un mayor esfuerzo en realizarlas.

Contar con herramientas de automatización de las pruebas resulta ser muy eficiente, sobre todo en aquellos proyectos donde es necesario ejecutar un número elevado de pruebas en un corto espacio de tiempo. Gracias a estas herramientas se ahorran grandes cantidades de tiempo y de dinero.

JUnit es un conjunto de bibliotecas utilizadas en Java para realizar y automatizar las pruebas unitarias de las aplicaciones.

Automatización de pruebas

La automatización de pruebas consiste en la utilización de software especializado que ejecuta las pruebas de manera controlada, presentando resultados y comparándolos con lo esperado.

Para alcanzar los resultados deseados de forma rápida se requiere un framework de prueba. JUnit es un framework para realizar y automatizar estas pruebas en Java, aunque no es el único, dependiendo del lenguaje podemos encontrar otros:

- **TestNG**. Creado para suplir algunas deficiencias en JUnit.
- **Mokito**. Parecido al anterior, muy usado en la industria y en otros frameworks como Spring.
- **PHPUnit**. Sistema para la realización de pruebas unitarias en PHP.
- **CPPUnit**. Sistema para la realización de pruebas unitarias en C/C++.

Características de JUnit

1. JUnit es un framework de código abierto que se utiliza para escribir y ejecutar pruebas unitarias.
2. Proporciona clases para la ejecución de pruebas o ejecutores de pruebas.
3. Proporciona anotaciones para identificar los métodos de ensayo o test.
4. Proporciona aserciones para resultados esperados del análisis.

Pruebas unitarias

La finalidad de una **prueba unitaria** es comprobar que un método funciona correctamente, es decir, hace lo que se espera que haga y no esté mal programado.

El único requisito para que una prueba unitaria resulte efectiva es que tenga únicamente dos posibilidades de resolución, prueba **correcta** o **incorrecta**. Para dar por buenas unas pruebas unitarias deben ser todas **correctas**

Una **prueba unitaria** se caracteriza por realizar una comparación del resultado esperado de un método con lo retornado por la codificación de dicho método. Esto es fundamental de entender para realizar unas pruebas buenas. Esto quiere decir que, para pasar las pruebas de un método, **debemos** de entender perfectamente que hace dicho método. Esto lo podemos hacer de dos maneras:

1. Leyendo la documentación del método. Evidentemente la documentación debe de estar bien hecha.
2. Entendiendo el código del método, para ello debemos de meternos dentro del método y entender que es lo que realmente hace.

Una vez entendido como funciona el método, el caso de prueba siempre se hará de la siguiente manera.

1. Plantearemos un **resultado esperado** bajo unos valores creados por nosotros, dicho resultado esperado debe ser igual al **resultado obtenido** después de ejecutar el método. Por ejemplo, imaginémonos un método sumar, que entendemos perfectamente que recibe dos números enteros, los suma y nos devuelve el resultado. En este caso crearemos dos valores, **4 y 5**, y entendemos que para que el método funcione bien, **el resultado esperado será el 9**.
2. Una vez establecido el resultado esperado, debemos ejecutar el método y sacar el resultado obtenido, y tal y como hemos avanzado antes, **el resultado esperado bajo nuestras condiciones debe de ser igual al resultado obtenido** para dar la prueba unitaria como **correcta**. Siguiendo el ejemplo anterior, si después de ejecutar el método con los valores 4 y 5 el resultado fuera 9, la prueba unitaria sería **correcta**, en caso contrario la prueba sería **incorrecta** y habría que revisar el código para ver qué es lo que falla.

Es muy importante entender que debemos de partir de resultados esperados, y que estos, **DEBEN SER IGUALES** a los resultados obtenidos. Esta es la base para pasar buenas pruebas.

Esta comparación de **resultado esperado** con **resultado obtenido** puede ser con cualquier tipo de dato, ya sea numérico, texto, booleano o incluso una excepción de programación esperada.

Para comparar los resultados esperados y los resultados obtenidos y facilitar nuestro trabajo, JUnit nos proporciona **aserciones**, métodos que nos ayudan a comparar datos y que veremos más adelante.

Alcance de las pruebas unitarias y cobertura de código

Cuando vamos a probar un método, es fundamental para dar la prueba por valida recorrer todas las líneas de método en cuestión y probar todos los posibles casos que vengan en la documentación. Probablemente para hacer esto, no nos valga con hacer una sola prueba, sino que tendremos que realizar varias, comúnmente llamado **batería de pruebas**.

Por ejemplo, supongamos que leyendo la documentación de un método nos dice lo siguiente

```
/**
* Método que comparar dos numero para ver cual de los dos es mayor
*
* @param numero1 primer número a comparar
```

```
* @param numero2 segundo número a comparar
* @return En caso de que el numero1 sea mayor que numero2 se devolverá un numero
positivo,
* en caso de que numero2 sea mayor que numero1 se devolverá un número negativo,
* en caso de que sean iguales se devolverá un 0
*/
public static int esMayor(int numero1, int numero2){
    //LOGICA DEL METODO
}
```

Supongamos también que no tenemos acceso al código fuente del método **esMayor**. En ese caso, para probarlo, tendríamos que basarnos en la documentación, prestando especial hincapié en el valor de retorno. Como podemos ver, tenemos 3 posibles valores de retorno, por lo cual tendremos que hacer **al menos tres pruebas** para dar por valida la prueba.

Una posible batería de pruebas sería la siguiente (que no única):

```
Prueba1.
    numero1 = 8
    numero2 = 4
    valor esperado = Un numero positivo (valor obtenido > 0)
Prueba2.
    numero1 = 5
    numero2 = 12
    valor esperado = Un numero negativo (valor obtenido < 0)
Prueba3.
    numero1 = 9
    numero2 = 9
    valor esperado = 0 (valor obtenido = 0)
```

Esto sería lo mínimo para dar por valida una prueba, probar todos los casos que vengan en la documentación o en su defecto, todos los casos o caminos que tengamos codificados en el método. Podríamos hacer más sin problema, pero al menos, todos los casos posibles.

JUnit nos ayudará es a crear esa batería de pruebas y a automatizarla, pero pensar las pruebas, los resultados esperados y los resultados obtenidos, correra de nuestra parte.

Como podemos apreciar, la documentación y las pruebas unitarias están bastante unidas. Si la documentación está bien hecha, podemos pasar las pruebas simplemente entendiéndola, no necesitaremos bucear en el código.

Con las pruebas unitarias se busca también dar un **porcentaje de cobertura** a tu código, es decir, por cuantas líneas, sobre el total de líneas que tiene tu programa, son capaces de pasar tus pruebas. Un 100% será el caso ideal, pero muchas veces es difícil de conseguir. Entre un 70% y un 90% suele ser bastante aceptable, pero depende de lo que pida nuestro responsable o el cliente.

Crear un módulo de pruebas JUnit5 con Eclipse

Lo primero que tenemos que hacer para pasar las pruebas unitarias de nuestra aplicación es crear un módulo donde declararlas. Para ello debemos de:

- Importar las librerías de JUnit a nuestro proyecto. JUnit5 no viene incluido en Java 11 por lo que hay que agregar las librerías al classpath. Además, JUnit 5 esta fragmentado en 3 proyectos, pero nosotros nos centraremos en **JUnit Jupiter**. Los otros dos serán **Junit vintage** con las características de JUnit4, y el otro **Junit platform**, principalmente para pruebas con la JVM.
- Crear una clase Java. Esta clase tendrá pruebas unitarias para ejecutar nuestra aplicación. Podemos tener tantas clases como queramos para pasar las pruebas.

Podemos ejecutar estas tareas de la siguiente manera:

1. Crear una carpeta al mismo nivel que src, podemos llamarla **test**
2. Agregar esa carpeta al classpath del proyecto
    - Botón derecho sobre la carpeta | build path | configure build path
    - Pulsamos en la pestaña source | add Folder | add carpeta test | OK
3. Creamos la clase que contendrá las pruebas unitarias. Para ello, botón derecho sobre la carpeta **test** | new | JUnit Test Case | new JUnit Jupiter test.
4. Se aparecerá una ventana, dentro de esta rellenamos el nombre del paquete (en este ejemplo, **junit5**) y el nombre de la clase (en este ejemplo **_00_Anotaciones**).
5. Si no tenemos JUnit5 agregado al proyecto, Eclipse nos preguntará para añadir el framework de JUnit5.
6. Podemos observar cómo se ha creado una librería en nuestro proyecto con el nombre de **JUnit 5**
7. A partir de aquí podemos crear todas las clases de JUnit que necesitemos para pasar las pruebas de nuestra aplicación.

Anotaciones en JUnit

Mediante las anotaciones en Java podemos agregar funcionalidad a nuestros programas de una manera sencilla y rápida.

En el caso de JUnit, las anotaciones irán encima de los métodos. Además, cada una de las anotaciones representará un momento del ciclo de vida de un caso de pruebas, ya que JUnit ejecutará todos los métodos que tengamos anotados.

Entre las anotaciones más importantes podemos encontrar:

- **@Test**, el principal y más importante ya que identifica un método como método test. Todos los métodos anotados con **@Test** se considerará una prueba unitaria que tiene que ser correcta.
- **@BeforeEach**, se ejecuta antes de cada test. Normalmente se utiliza para preparar el entorno de testing (por ejemplo: inicialización de clases o lectura de datos de entrada).
- **@AfterEach**, se ejecuta después de cada test. Normalmente se utiliza para limpiar el entorno de testing.
- **@BeforeAll**, se ejecuta una vez, antes de comenzar todos los tests. Este método debe definirse como static.
- **@AfterAll**, se ejecuta una vez, cuando los tests han finalizado. Este método deben definirse como static para trabajar con JUnit.

NO es necesario crear la clase de pruebas con todas las anotaciones, pero al menos debe haber un método anotado con **@Test**.

Aserciones en JUnit

Las aserciones nos permiten verificar el valor del resultado esperado con el valor de retorno obtenido del metodo probado. Son funciones de aceptación y fundamentales para trabajar con JUnit. Las aserciones se colocan dentro de los métodos etiquetados con @Test, es decir, métodos de prueba. Si la aserción se cumple, la prueba se dará por valida, si la aserción no se cumple, la prueba fallará.

Una prueba puede tener muchas aserciones, pero en cuanto una aserción no se cumpla, la prueba se dara por fallada y se dejará de ejecutar el método de la prueba.

Las aserciones van dentro de los métodos de pruebas y podemos poner todas las aserciones que se necesiten para pasar la prueba.

JUnit proporciona varios métodos de aserción para escribir pruebas de JUnit, las más importantes los ponemos en negrita:

- **assertFalse(boolean condición)**. Verifica si la condición es falsa
- **assertTrue(boolean condition)**. Verifica si la condición es verdadera
- **assertEquals(valor_esperado , valor_obtenerido)**. Comprueba si los valores son iguales. Los valores pueden ser objetos o primitivos. Si son objetos, la igualdad se hará por el método **equals**
- **assertNotEquals(valor_esperado , valor_obtenerido)**. Comprueba si los valores **NO** son iguales
- **assertNull(java.lang.Object obtenido)**. Comprueba que la referencia sea nula.
- **assertNotNull(java.lang.Object obtenido)**. Comprueba que la referencia **NO** sea nula.
- assertSame(java.lang.Object expected, java.lang.Object obtenido). Comprueba que ambas referencias apuntan al mismo objeto.
- assertNotSame(java.lang.Object expected, java.lang.Object obtenido). Comprueba que ambas referencias **NO** apuntan al mismo objeto.
- assertArrayEquals(Array1, Array2). Comprueba que ambos arrays tengan los mismos elementos.
- assertThrows(Exception, función lambda). Comprueba a partir de una función lambda si ha ocurrido una excepción específica.

Ademas de las aserciones, JUnit nos proporciona algún método adicional.

- **fail()**, si se llega a ejecutar este método, la prueba fallará. Suele ir dentro de condicionales.

Ejecución de una clase de pruebas en Eclipse

Pulsaremos el botón derecho sobre la clase que queremos ejecutar | run as | JUnit Test.

Se abrirá una pestaña nueva **JUnit** donde se presentarán los resultados en función de los colores mostrados:

1. **Verde**, todas las pruebas han ido bien, es decir, los resultados esperados eran iguales a los resultados obtenidos. Dicho de otra manera, todas las aserciones se han cumplido y estamos en el estado ideal.
2. **Rojo**, alguna prueba ha fallado, se debe de revisar la prueba y el método probado.

Para dar por satisfactorias las pruebas debe de salir el color **verde** al ejecutarlas.

También podemos ver en detalle los métodos de test lanzados, aquí podemos encontrar 3 colores.

1. **Verde**, la prueba ha ido bien.

2. **Azul**, la prueba ha fallado, es decir, alguna aserción NO se ha cumplido o se ha ejecutado la función **fail()**.
3. **Rojo**, ha habido algún tipo de error o excepción en el test. En este caso no es concluyente el resultado y se debería revisar la prueba.

Tips de desarrollo

Unos buenos pasos para desarrollar código son los siguientes:

1. Definir la clase y documentarla
2. Definir los atributos de clase y documentarlos (si es necesario).
3. Definir los métodos a medida que los vayamos necesitando, y antes de codificarlos, realizar su documentación.
4. Implementar el método basandose en la documentación hecha. Podriamos cambiar la documentación si lo vemos necesario.
5. Realizar la prueba unitaria (JUnit) del método.
6. En caso de que falle la prueba, debermos de arreglar el códido. Volvemos al punto 5.
7. Es posible que al pasar la prueba, nos veamos en la necesidad de volver a cambiar la documentación hecha, en el caso de darnos cuenta de algunos casos que no contemplamos.

Ejemplos JUnit

Dentro de este proyecto, en la carpeta **test/junit5** podemos encontrar ejemplos numerados sobre JUnit.

Bibliografía

- https://junit.org/junit5/