PRUEBAS UNITARIAS. JUNIT

CODE UD7 – Tema 1

IES Plurilingüe Antón Losada Diéguez

Adrián Fernández González

Tabla de contenido

Pruebas unitarias. JUnit

1. Introducción

Una prueba unitaria es aquella diseñada para analizar una unidad de código. En programación funcional, esta unidad sería un procedimiento o función, mientras que en orientación a objetos sería una clase o un prototipo dependiendo del enfoque.

Cada lenguaje dispone de sus utilidades para realizar las pruebas, en este caso, JUnit es el framework para la realización de las pruebas en Java.

2. Ventajas del uso de pruebas unitarias

Al analizar cada parte del sistema de forma independiente, las pruebas unitarias proporcionan una serie de ventajas clave:

Fomentan el cambio: Al probar cada parte de forma independiente, permiten que el desarrollador cambie el funcionamiento interno de cada parte y tenga una evaluación inmediata de su correcto funcionamiento. En otras palabras, favorece la refactorización.

Simplifican la integración: Dado que cada parte del sistema está siendo evaluado continuamente, facilita el proceso de integración, ya que solo sería necesario probar la interacción entre las partes.

Documentan el código: A la hora de documentar el código, las pruebas sirven como ejemplo del uso de cada parte del sistema.

Acotan los errores: Al hacer pruebas de cada parte, permite detectar con mayor precisión el lugar en el que se produce un error.

3. Limitaciones

Una de las principales limitaciones de las pruebas unitarias es que, al probar de forma independiente cada parte del sistema, no permite detectar los fallos de interacción entre las mismas o los fallos de integración del sistema con el hardware y el software del entorno final.

Por otro lado, a veces se hace difícil saber que entradas podría recibir una parte del sistema, por lo que la efectividad de la prueba se limita al nivel de similitud con la realidad que tengan las entradas.

Otro problema es que ciertas estructuras del código no pueden ser evaluados más allá de su ejecución completa, ya que habría que controlar valores internos. Un claro ejemplo de esto son los bucles, que tendrían que ser aislados.

Por todo esto, las pruebas unitarias suelen combinarse con otro tipo de pruebas para probar la totalidad del sistema.

4. Diseño de la prueba

Antes de empezar a programar la prueba, es necesario diseñarla siguiendo las siguientes características:

Automatizables: Que se ejecuten de forma automática para que, cuando se realice cualquier tipo de cambio, se pasen las pruebas y verifique que todo funciona correctamente.

Completas: Deben probar todas las partes del sistema. A mayor código probado, mayor probabilidad de detección de fallos.

Rápidas: Deben poder ejecutarse en fracciones de segundo, de lo contrario retrasarían el desarrollo y podrían ser deshabilitadas manualmente, dejando partes del sistema sin probar.

Reutilizables: Las pruebas deberían ser reutilizables, poder usarlas una y otra vez aun cuando el código interno cambie. Esto ahorra tiempo de desarrollo, ya que, si hubiese que crear pruebas cada vez que algo cambia, se gastaría demasiado tiempo.

Independientes: El resultado de cada prueba tiene sentido por sí mismo y no depende del resultado de otras pruebas.

Profesionales: El código de las pruebas ha de ser tratado como cualquier otro del sistema, ha de ser eficiente, estar documentado, seguir las reglas de la empresa, etc.

Normalmente se diseñan siguiendo algún tipo de estrategia, por ejemplo, el camino básico de las pruebas de caja blanca o la partición de equivalencia de caja negra, entre otros.

5. JUnit

Tal como se mencionó antes, las pruebas unitarias en Java se realizan con JUnit, estándar que viene incluido en los IDE.

Para crear las pruebas sobre un sistema, se crea un nuevo tipo de elemento en el proyecto, al igual que una clase u otro tipo de archivo, denominado JUnit Test.

Habitualmente, se crea un archivo de pruebas por cada clase a probar y un método test por cada valor a probar de un método en concreto.

La estructura base de los test es siempre la misma, un método *public void* con una anotación *@Test* indicando que el método es una prueba. Habitualmente se suele nombrar como test, el nombre del método y el caso a probar.

En su interior el procedimiento suele ser siempre el mismo:

1. Se crea una instancia de la clase a probar. Esta puede ser creada en el propio método de prueba o como atributo de la clase de pruebas.
2. Se llama al método a probar pasando los parámetros pertinentes y recogiendo su valor.
3. Se estipula el valor esperado y se compara con resultado de la ejecución mediante uno de los métodos *assert* de JUnit.

```java
public int prueba(int a, int b)
{
    int ret = 0;
    if(a > b)
    {
        ret = -1;
    }else{
        if(a < b)
        {
            ret = 1;
        }
    }
    return ret;
}
```

```java
@Test
public void testPruebaA()
{
    MiClase instance = new MiClase();
    int a = 1;
    int b = 0;
    int expected = -1;
    int result = instance.prueba(a, b);
    assertEquals(expected, result);
}

@Test
public void testPruebaB()
{
    MiClase instance = new MiClase();
    int a = 0;
    int b = 1;
    assertEquals(1, instance.prueba(a, b));
}

@Test
public void testPruebaEq()
{
    MiClase instance = new MiClase();
    assertEquals(0, instance.prueba(1, 1));
}
```

En este ejemplo se han creado tres pruebas para verificar que funciona correctamente en las tres casuísticas posibles, que A sea mayor que B, que B sea mayor que A o que A y B sean iguales.

5.1. Métodos assert

JUnit dispone de cinco métodos *assert* para la verificación de los resultados:

assertEquals(esperado, actual): Verifica si ambos valores son iguales.

assertTrue(boolean condition): Verifica si una condición devuelve true.

assertFalse(boolean condition): Verifica si una condición devuelve false.

assertNull(Object obj): Verifica si un objeto es *null*.

assertNotNull(Object obj): Verifica si un objeto NO es *null*.

5.2. Anotaciones

Además de la anotación ***@Test*** que indica que el método es una prueba, JUnit dispone de otras para realizar antes y después de las pruebas. A la izquierda, para versiones de JUnit 4 o inferiores, a la derecha para versiones de JUnit 5+ (Jupiter).

@Test(timeout=1000) Especifica que la prueba no puede tardar más de 1000 milisegundos (1 segundo) en ejerutarse. Útil para pruebas de eficiencia.

@BeforeClass / @BeforeAll El método se ejecutará antes de ejecutar el conjunto de pruebas. Se suele utilizar para inicializar conexiones o servicios para las pruebas como la conexión a una base de datos o las tablas a utilizar. El nombre por defecto del método es *setUpBeforeClass*().

@AfterClass / @AfterAll El método se ejecutará después de ejecutar todas las pruebas. Suele utilizarse para cerrar conexiones o servicios abiertos anteriormente. El nombre por defecto del método es *tearDownAfterClass*().

@Before / @BeforeEach El método se ejecutará antes de ejecutar cada una de las pruebas. Habitualmente se usa para inicializar los objetos a utilizar, como la instancia de la clase a probar. El nombre por defecto del método es *setUp*().

@After / @AfterEach El método se ejecutará después de ejecutar cada una de las pruebas. El nombre por defecto del método es *tearDown*().

5.3. Principio de independencia

Hay que tener en cuenta que JUnit fue diseñado para cumplir con el principio de independencia de las pruebas unitarias, por lo que la clase que genera las pruebas crea una instancia nueva de sí misma cada vez que ejecuta uno de sus métodos.

Esto quiere decir que, si se crea una instancia de un objeto como atributo y se modifica en uno de sus métodos de prueba, cuando se ejecute otro método, la instancia del objeto se ha reiniciado, como si se hubiese creado de nuevo.

Por tanto, la instancia se puede crear como un atributo o como una variable en cada método, ya que su estado será siempre el inicial en ambos casos.

5.3.1. Inicializar el objeto a utilizar

Pese a que se pueda crear el objeto en cualquier parte, lo recomendable es declararlo como atributo y luego inicializarlo en el método *@Before / @BeforeEach*. Esto es así porque este método está gestionado por Junit y, en caso de algún fallo facilita una información mayor que una excepción normal.

5.3.2. Inicializar servicios y conexiones

Dado que las conexiones y los servicios como bases de datos, ficheros, etc. se van a utilizar en todas las pruebas, lo habitual es inicializarlos en el método *@BeforeClass / @BeforeAll* y cerrarlos en *@AfterClass / @AfterAll*.