

## 7. PRUEBAS UNITARIAS

Las pruebas unitarias, o prueba de la unidad, tienen por objetivo probar el correcto funcionamiento de un módulo de código. El fin que se persigue, es que cada módulo funciona correctamente por separado.

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



Una unidad es la parte de la aplicación más pequeña que se puede probar. En programación procedural, una unidad puede ser una función o procedimiento, En programación orientada a objetos, una unidad es normalmente un método.


Con las pruebas unitarias se debe probar todas las funciones o métodos no triviales de forma que cada caso de prueba sea independiente del resto.

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.
- **Repetibles o 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.

El objetivo de las pruebas unitarias es aislar cada parte del programa y demostrar que las partes individuales son correctas. Las pruebas individuales nos proporcionan cinco ventajas básicas:

1. **Fomentan el cambio:** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifica la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
3. **Documenta el código:** Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la implementación:** Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
5. **Los errores están más acotados y son más fáciles de localizar:** dado que tenemos pruebas unitarias que pueden desenmascararlos.

 <p>CIFP VIRGEN DE GRACIA</p>	<p align="center"><b>UT3-DISEÑO Y REALIZACIÓN DE PRUEBAS</b>  <b>Entornos de desarrollo (1º DAW)</b>  <b>Pruebas unitarias _ JUnit</b></p>	<p align="center"><b>Dpto. INFORMÁTICA</b>   Curso: 2022-23</p>
--	--	---

### 7.1. Herramientas para Java.

Entre las herramientas que nos podemos encontrar en el mercado, para poder realizar las pruebas, las más destacadas serían:

#### **Jtiger:**


- ✓ Framework de pruebas unitarias para Java.
- ✓ Es de código abierto.
- ✓ Capacidad para exportar informes en HTML, XML o texto plano.
- ✓ Es posible ejecutar casos de prueba de Junit mediante un plugin.
- ✓ Posee una completa variedad de aserciones como la comprobación de cumplimiento del contrato en un método.
- ✓ Los metadatos (*Conjunto de datos que se utilizan para describir otros datos*) de los casos de prueba son especificados como anotaciones del lenguaje Java
- ✓ Incluye una tarea de *Ant* para automatizar las pruebas.
- ✓ Documentación muy completa en JavaDoc, y una página web con toda la información necesaria para comprender su uso, y utilizarlo con IDE como Eclipse.
- ✓ El Framework (*Estructura conceptual y tecnológica de soporte definida, normalmente con módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado*) incluye pruebas unitarias sobre sí mismo.

#### **TestNG:**

- ✓ Está inspirado en JUnit y NUnit.
- ✓ Está diseñado para cubrir todo tipo de pruebas, no solo las unitarias, sino también las funcionales, las de integración ...
- ✓ Utiliza las anotaciones de Java 1.5 (desde mucho antes que Junit).
- ✓ Es compatible con pruebas de Junit.
- ✓ Soporte para el paso de parámetros a los métodos de pruebas.
- ✓ Permite la distribución de pruebas en máquinas esclavas.
- ✓ Soportado por gran variedad de plug-ins (Eclipse, NetBeans, IDEA ...)
- ✓ Las clases de pruebas no necesitan implementar ninguna interfaz ni extender ninguna otra clase.
- ✓ Una vez compiladas la pruebas, estas se pueden invocar desde la línea de comandos con una tarea de Ant o con un fichero XML.
- ✓ Los métodos de prueba se organizan en grupos (un método puede pertenecer a uno o varios grupos).

#### **JUnit:**

- ✓ Framework de pruebas unitarias creado por Erich Gamma y Kent Beck.
- ✓ Es una herramienta de código abierto.
- ✓ Multitud de documentación y ejemplos en la web.
- ✓ Se ha convertido en el estándar de hecho para las pruebas unitarias en Java.
- ✓ Soportado por la mayoría de los IDE como eclipse, Netbeans, VisualStudioCode o IntelliJ.
- ✓ Es una implementación de la arquitectura xUnit para los frameworks de pruebas unitarias.

 <p>CIFP VIRGEN DE GRACIA</p>	<p align="center"><b>UT3-DISEÑO Y REALIZACIÓN DE PRUEBAS</b>  <b>Entornos de desarrollo (1º DAW)</b>  <b>Pruebas unitarias _ JUnit</b></p>	<p align="center"><b>Dpto. INFORMÁTICA</b>   Curso: 2022-23</p>
--	--	---

- ✓ Posee una comunidad mucho mayor que el resto de los frameworks de pruebas en Java.
- ✓ Soporta múltiples tipos de aserciones.
- ✓ Posibilidad de crear informes en HTML.
- ✓ Organización de las pruebas en Suites de pruebas.
- ✓ Es la herramienta de pruebas más extendida para el lenguaje Java.

## 7.2 PRUEBAS con Junit

JUnit es un framework desarrollada para poder probar el funcionamiento de las clases y métodos que componen nuestra aplicación, y asegurarnos de que se comportan como deben ante distintas situaciones de entrada.

Para ello, habrá que identificar los casos de uso, datos de entrada y resultados esperados con las técnicas de caja blanca y negra ya conocidos. Una vez esté disponible esta información, toca lanzar con JUnit las ejecuciones programadas y valorar si los resultados obtenidos son exitosos.

### CONCEPTO FUNDAMENTAL.

En esta herramienta el concepto fundamental es:

- **CASO DE PRUEBA** (TEST CASE). Son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o.
- **SUIT DE PRUEBA** (TEST SUIT). Podremos organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados.

Conceptos relacionados con JUnit en su versión 5:

- Etiquetas disponibles para configurar pruebas en Java (**anotaciones**).
- Llamadas a métodos de prueba para comprobar el funcionamiento del código (**assertions**).
- Guía de uso de JUnit 5: <https://junit.org/junit5/docs/current/user-guide/>

Al generar un test de una clase, nos crea automáticamente una clase que se llamara, como la clase pero terminado en test, con el siguiente formato:

```
@BeforeClass
public static
void setUpClass
(){
}
@AfterClass
public static
void
tearDownClass(){
}
@Before
public static
void setUp(){
}
@After
public static
void tearDown(){
}
```

Estas serían las etiquetas (*anotaciones*), existen los inicializadores y finalizadores, que se ejecutarán antes y después de ciertos eventos las más utilizadas:

#### Inicializador de la clase

- **@BeforeClass:** Sólo puede haber un método con este marcador, es invocado una vez al principio de todos los test. Se suele usar para inicializar atributos. Se puede utilizar, para crear una conexión a una base de datos o antes de ejecutar cualquier método, etc..


#### Finalizador de la clase

- **@AfterClass:** Indica el método finalizador de la clase. Solamente se ejecutará una vez justo después de la ejecución de los métodos de la clase test.

Ej: si se necesita desconectarse de una base de datos.

#### Inicializador del test

- **@Before:** Marca el método que va a ejecutarse antes de cada test. No es un método necesario. Se ejecutará antes de cada test y si, por ejemplo, quieren inicializarse variables, este método sería el adecuado para hacerlo.

 CIFP VIRGEN DE GRACIA	<p style="text-align: center;"> <b>UT3-DISEÑO Y REALIZACIÓN DE PRUEBAS</b>  <b>Entornos de desarrollo (1º DAW)</b>  <b>Pruebas unitarias _ JUnit</b> </p>	<p style="text-align: center;"> <b>Dpto. INFORMÁTICA</b>           Curso: 2022-23       </p>
--	---	--

### Finalizador del test

- **@After:** Se ejecuta después de cada test. No es un método necesario. Se ejecutará después de cada test y, por ejemplo, puede utilizarse para limpiar variables o estructuras de datos utilizadas en los test

### Otras

- **@Ignore:** Los métodos marcados con esta anotación no serán ejecutados.
- **@Test:** Identifica los métodos test que deben ser probados

```

@Ignore
@Test
public void testMultiplica() {
    System.out.println("multiplica");
}

```

### clase Assertions

JUnit utiliza la clase **Assertions** para lanzar los test, que básicamente está compuesta por una serie de métodos, que una vez llamados ejecutan los métodos a probar y analizan su comportamiento comparándolos con los resultados que se espera de ellos.

Así, hay métodos que nos permiten comprobar si dos valores son o no iguales, si el valor del parámetro pasado se puede resolver como true o false, si el tiempo consumido en ejecutar un método supera el previsto, etc.

Ahora vamos a ver los métodos que usaremos, disponibles en JUnit, todos se invocan con la **clase assert** de forma estático. Los más usados son:

**assertEquals(resultado esperado, resultado actual):** le pasamos el resultado que nosotros esperamos e invocamos la función que estamos testando.

**assertNull(objeto):** si un objeto es null el test será exitoso.

**assertNotNull(objeto):** al contrario que el anterior.

**assertTrue(condición):** si la condición pasada (puede ser una función que devuelva un booleano) es verdadera el test será exitoso.

**assertFalse(condición):** si la condición pasada (puede ser una función que devuelva un booleano) es falsa el test será exitoso.

**assertSame(Objeto1, objeto2):** compara las referencias de los objetos.

**assertNotSame(Objeto1, objeto2):** al contrario que el anterior.

Para entender mejor los métodos assert, se va a modificar la clase **ClaseUnoJUnitTest** nuevamente, sustituyendo los mensajes por pantalla que se mostraban en la sección anterior por llamadas assert que permitan probar los métodos de la clase ClaseUnoJUnit\_.

## Ejemplo Mi primera clase de test

### Crear una primera clase de test \_ Netbeans

El primer paso para crear una clase de test es tener una primera clase por testear.

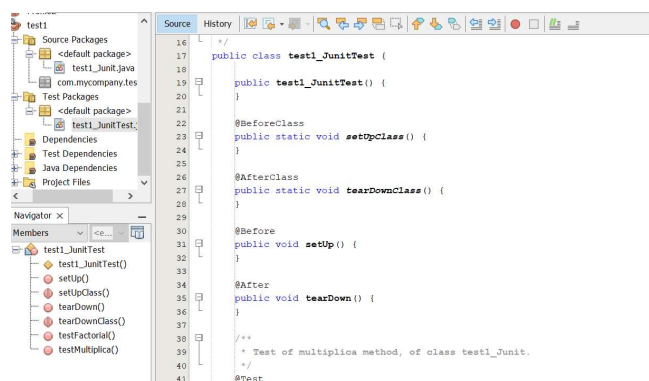
```
public class Test_Junit {
    /**
     * @param args the command line argument
     */
    public int multiplica (int a, int b){
        return a*b;
    }
    public int factorial (int numero){
        int factorial = numero;
        for (int i = (numero -1); i>1 ; i--){
            factorial = factorial * i;
        }
        return factorial;
    }
}
```

Esta clase tiene dos métodos: uno llamado multiplica y otro llamado factorial.

El siguiente paso es elaborar una clase de prueba para el código creado. Para ello, se hace clic con el botón derecho del ratón y, sobre el menú desplegable, se elige **Tools->Create Test**.

Para la creación de test, se ha elegido Junit 4. Hay que tener en cuenta que la estructura de los test en Junit 3 y 4 son diferentes.

Se creará la clase **Test\_JunitTest.java**, la cual tendrá un pequeño esqueleto que servirá en un futuro para realizar los test de la clase creada.



```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
```



```
/**
 *
 * @author Garcia
 */
public class test1_JunitTest {

    public test1_JunitTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of multiplica method, of class test1_Junit.
     */
    @Test
    public void testMultiplica() {
        System.out.println("multiplica");
        int a = 0;
        int b = 0;
        test1_Junit instance = new test1_Junit();
        int expResult = 0;
        int result = instance.multiplica(a, b);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

    /**
     * Test of factorial method, of class test1_Junit.
     */
    @Test
    public void testFactorial() {
        System.out.println("factorial");
        int numero = 0;
        test1_Junit instance = new test1_Junit();
        int expResult = 0;
        int result = instance.factorial(numero);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
    }
}
```

```
fail("The test case is a prototype.");  
}  
  
}
```

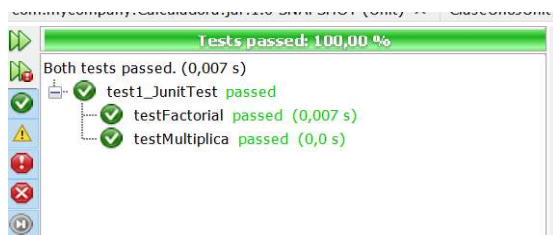
#### NOTA

Para este test, se ha utilizado el método **asserEquals** de Junit. Esta afirmación funciona pasándole el resultado esperado y unas variables determinadas. Cuando se realiza el test, Junit comprueba que el resultado esperado es el mismo que han devuelto las variables cuando se ejecuta el método.

Pueden colocarse tantos **assertEquals** en el código como sea necesario.

Tenemos que cambiar los valores de las variables y comentar la línea de **//fail**

Pulsando sobre la clase por testear y eligiendo la opción **Test**, se lanza automáticamente el test creado. Dependiendo de los *asserts* que se establezcan, el test fallará o pasará con éxito.

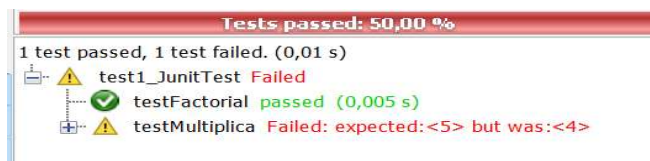


Si se modifica el test **testMultiplica** de tal manera que se altera el método como puede observarse:

```
public void testMultiplica() {  
    System.out.println("multiplica");  
    int a = 2;  
    int b = 2;  
    Test1 instance = new Test1();  
    int expResult = 5;  
    int result = instance.multiplica(a,  
    b);  
    assertEquals(expResult, result);  
}
```

El resultado al ejecutar el test será el siguiente:





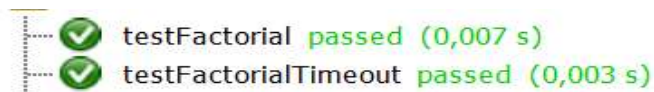
Supuestamente, el test del método multiplica espera un resultado de 5, pero el resultado ha sido 4. En ese caso, el test automatizado detecta un fallo e informa de ello.

### Test de rendimiento

Se realiza el test de rendimiento del método factorial. Se incluirá un test para que el factorial de 300 tarde menos de 1 segundo. El test es el siguiente:

```
@Test(timeout=1000)
public void testFactorialTimeout() {
    System.out.println("factorial con
    timeout");
    int numero = 300;
    Test1 instance = new Test1();
    int result = instance.factorial
    (numero);
}
```

Se ejecuta y se observa que el código pasa satisfactoriamente el test, puesto que su ejecución no supera el segundo



### Deshabilitar un test

Deshabilitar el test de multiplicación, colocando @Ignore en una línea antes del @Test, con lo que JUnit ignora su realización. – como resultado de su ejecución será el siguiente:

