

**MP\_0487. Entornos de desarrollo**

**UF3. Diseño y realización de pruebas**

**3.2. Herramientas de testeo  
en los entornos de desarrollo**

# Índice

---

|   |                           |    |
|---|---------------------------|----|
| ☰ | Objetivos                 | 3  |
| ☰ | Banco de pruebas          | 4  |
| ☰ | Automatización de pruebas | 7  |
| ☰ | Pruebas unitarias         | 9  |
| ☰ | JUnit                     | 10 |
| ☰ | La clase JUnit            | 14 |
| ☰ | Métodos JUnit             | 16 |
| ☰ | Anotaciones JUnit         | 19 |
| ☰ | Suites de pruebas         | 21 |
| ☰ | Ejemplo de uso            | 22 |
| ☰ | Pruebas parametrizadas    | 27 |
| ☰ | Resumen                   | 28 |

# Objetivos

---

En esta lección perseguimos los siguientes objetivos:

- 1 Conocer las características de la automatización de pruebas.
  - 2 Trabajar en los ámbitos de aplicación de las pruebas y su automatización.
  - 3 Conocer cómo se pueden automatizar las pruebas.
  - 4 Conocer cuáles son las herramientas más usadas en la automatización de pruebas.
  - 5 Conocer cuáles son los estándares sobre pruebas de software.
- 

¡Ánimo y adelante!

## Banco de pruebas

---

Las tareas de realización de pruebas se pueden resolver de una forma automatizada siempre que contemos con herramientas que lo permitan.

Las pruebas constituyen **una fase costosa y laboriosa** dentro del proceso de desarrollo del software. Contar con herramientas de automatización de las pruebas es algo cada vez más frecuente.

Las **automatización** resulta ser muy eficiente, por ejemplo en pruebas de verificación de funcionalidades clave o en pruebas de regresión, donde es necesario ejecutar un número elevado de pruebas en un corto espacio de tiempo.

Las **herramientas de pruebas** de software automatizadas ya han sido utilizadas en aplicaciones de escritorio, pero en general siempre eran aplicaciones muy complejas asociadas a licencias comerciales de alto coste.

Sin embargo, la automatización de la verificación y las pruebas de las interfaces basadas en web son más sencillas de implementar. En la actualidad existe un gran número de aplicaciones de software libre que pueden ser utilizadas.

Se denomina banco de pruebas del software a un conjunto integrado de herramientas que soportan el proceso de pruebas. Entre estas herramientas se pueden citar:

### **Gestor de pruebas**

Como su nombre indica, es capaz de gestionar la ejecución de las pruebas del programa y mantiene un registro de los datos de entrada de las pruebas, los resultados esperados y aspectos del programa que han sido probados.

Un ejemplo de gestor de pruebas es JUnit, que permite automatizar las pruebas unitarias de una aplicación Java. JUnit se compone de un conjunto de bibliotecas Java que permite probar si los métodos implementados en las clases del código se comportan de manera correcta. Para ello, a partir de valores de entrada, se evalúan los datos de salida obtenidos y, si no son los esperados, devolverá un mensaje de fallo.

### **Analizador dinámico**

Esta herramienta incorpora un código dentro del programa con el objetivo de contabilizar el número de veces que se ejecuta cada sentencia.

### **Oráculo**

Es una herramienta que obtiene las predicciones de los resultados que se generan a partir de los datos de prueba. Esta herramienta puede ser la misma aplicación de una versión anterior o bien un prototipo creado para este fin.

### **Comparador de ficheros**

Herramienta que compara los resultados obtenidos con otros de pruebas previas. Se utiliza en pruebas de regresión, donde se comparan pruebas de distintas versiones.

### **Generador de informes**

Proporciona la definición de informes para los resultados de las pruebas.

### **Generador de datos de pruebas**

Genera los datos de pruebas para el programa que se desea probar. Estos se pueden obtener desde una BBDD o bien de forma aleatoria.

## Simulador

Pueden ser, por ejemplo, pruebas de interfaces de usuario, que son programas guiados por scripts que simulan numerosas interacciones de los usuarios.

 Debido al gran coste que supone implementar un banco de pruebas, estos solo se implementan de forma completa para grandes desarrollos de software.

# 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.

Tipos de pruebas que permite el software:

## Pruebas codificadas

Se automatizan las pruebas unitarias utilizando casos de uso.

## Pruebas de usuario

Se graban acciones que realizarán los usuarios sobre la interfaz de la aplicación a evaluar. Con esto se consigue que las acciones repetitivas puedan ser ejecutadas las veces que haga falta.

Para **automatizar** el proceso de desarrollo de las pruebas se utilizan *frameworks* como:

### JUnit 5

JUnit 5 es la próxima generación de JUnit. El objetivo es crear una base actualizada para las pruebas del desarrollador en la JVM. Esto incluye centrarse en Java 8 y superior, así como habilitar muchos estilos diferentes de prueba.

JUNIT EN JAVA

## NUnit

NUnit es un marco de pruebas unitarias para todos los lenguajes .NET. Inicialmente portado desde JUnit , la versión de producción actual, versión 3, ha sido completamente reescrito con muchas características nuevas y soporte para una amplia gama de plataformas .NET.

NUNIT EN .NET

Existen muchos otros entornos de pruebas dependiendo del lenguaje de programación que utilicemos, a continuación nombramos algunos de ellos:

- **TestNG.** Creado para suplir algunas deficiencias en JUnit.
- **JTiger.** Basado en anotaciones, como TestNG, también para Java.
- **SimpleTest.** Entorno de pruebas para aplicaciones realizadas en PHP.
- **PHPUnit.** Sistema para la realización de pruebas unitarias en PHP.
- **CPPUnit.** Versión del *framework* para lenguajes C/C++.
- **FoxUnit.** Entorno de pruebas para Microsoft Visual FoxPro.
- **MOQ.** Entorno para la creación dinámica de objetos simuladores (*mocks*).
- **QUnit.** Librería para pruebas unitarias en JavaScript. Creada por la fundación jQuery.

# Pruebas unitarias

---

Las pruebas unitarias son un tipo de prueba que podemos realizar a nuestro software.

Un caso de prueba unitaria es una parte de código que garantiza que la otra parte del código (método) funciona como se esperaba.

Para alcanzar los resultados deseados de forma rápida se requiere un *framework* de prueba. JUnit es un *framework* bastante bueno para realizar y automatizar estas pruebas en Java.

Una prueba unitaria se caracteriza por realizar una comparación del resultado esperado de un método con lo **retornado por la codificación** de dicho método.

Esta comparación puede ser con cualquier tipo de dato o conjunto de datos, ya sean numéricos, textos, booleanos o incluso una excepción de programación esperada. El único requisito para que una prueba unitaria resulte efectiva es que tenga únicamente dos posibilidades de resolución (correcta/incorrecta).

Con las pruebas unitarias podemos comparar el resultado aislado de un método, como el cálculo de una fórmula matemática que nos retorne un único dato estático. Pero también podemos realizar pruebas enviando una serie de datos parametrizados, sometiendo al método a realizar operaciones con diferentes rangos y así comprobar su correcto funcionamiento en diferentes situaciones.

## JUnit

---

JUnit es un *framework* de pruebas unitarias para el lenguaje de programación Java. Es importante para el desarrollo de las pruebas conducidas y es uno de una familia de *frameworks* de pruebas unitarias conocidos colectivamente como xUnit.

JUnit es un *framework* de pruebas de regresión utilizado por los desarrolladores para implementar pruebas unitarias en Java, acelerar la velocidad de programación y aumentar la calidad del código.

JUnit promueve la idea de "**primera prueba después de la codificación**", que pone énfasis en la creación de datos de prueba para una pieza de código que puede ser probado por primera vez y luego pueda ser implementado. Este enfoque se basa en "poner a prueba un poco, codificar un poco, prueba un poco, codifica un poco...", lo que aumenta la productividad del programador, la estabilidad del código del programa y reduce el estrés del programador y del tiempo dedicado a la depuración.

El *framework* JUnit se puede integrar fácilmente con cualquiera de los siguientes entornos: NetBeans, Eclipse, Ant y Maven.

Las ventajas principales de JUnit son:

- 1 JUnit es un *framework* de código abierto que se utiliza para escribir y ejecutar pruebas.
- 2 Proporciona una anotación para identificar los métodos de ensayo.
- 3 Proporciona aserciones para resultados esperados del análisis.
- 4 Proporciona **clases para la ejecución** de pruebas o ejecutores de pruebas.
- 5 Las pruebas JUnit permiten escribir código más rápido, incrementando la calidad.
- 6 JUnit es **sencillo y elegante**. Es menos complejo y requiere menos tiempo.
- 7 Las pruebas JUnit se pueden **ejecutar de forma automática** y verifican sus propios resultados además de proporcionar una retroalimentación inmediata. No hay necesidad de crear ningún informe de resultados de la prueba de manera manual.
- 8 Las pruebas JUnit pueden ser **organizadas en conjuntos de pruebas** que contienen los casos de prueba e incluso de otros conjuntos de pruebas.
- 9 JUnit **muestra el progreso de la prueba** en una barra que es de color verde si la prueba va bien y se vuelve roja cuando falla una prueba.

## Las características de JUnit

### Clases de JUnit

Las clases de JUnit son clases importantes que se utilizan para escribir y probar JUnits.

Algunas de las clases importantes son:

- **Assert**: contienen un conjunto de métodos de aserción.
- **TestCase**: contienen un caso de prueba definido como accesorio para ejecutar varias pruebas.
- **TestResult**: contienen métodos para recoger los resultados de la ejecución de un caso de prueba.

## Bancos de pruebas o “test suites”

El banco de pruebas o “*test suites*” son un envoltorio de unos pocos casos de pruebas funcionales que se ejecutarán juntas. En JUnit las anotaciones `@RunWith` y `@Suite` se utilizan para ejecutar la prueba en conjunto.

## Ejecutor de prueba

Se utiliza para ejecutar los casos de prueba.

## ¿Cómo funciona JUnit?

JUnit se encarga de ejecutar todos los métodos `testxxx()` que contiene la clase.

En el siguiente ejemplo el único método con este nombre es `testConcat()`, por tanto JUnit lo ejecutará. Dentro de `testConcat()` hay una llamada al método `assertTrue()`. Este método comprueba si la expresión que recibe como argumento es cierta, y trasmite el resultado a JUnit.

```
import junit.framework.*;
//Un test de ejemplo sobre la clase String.
public class EjemploTest extends TestCase {
    public void testConcat() {
        String s = "hola";
        String s2 = s.concat(" que tal");
        assertTrue(s2.equals("hola que tal"));
    }
    public static Test suite() {
        return new TestSuite(EjemploTest.class);
    }
    public static void main (String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Ejemplo clase JUnit.

Para conocer más a fondo el uso de JUnit, tanto en NetBeans como en Eclipse, y su funcionamiento básico, te recomendamos las siguientes lecturas:

**Tutorial JUnit (Eclipse)**

JUnit es una biblioteca de Java que te ayuda a realizar pruebas unitarias.

**TUTORIAL JUNIT****Tutorial JUnit (NetBeans)**

Este tutorial presenta los conceptos básicos de escritura y ejecución de pruebas de unidad JUnit en el IDE NetBeans.

**TUTORIAL JUNIT**

## La clase JUnit

Como todo en Java, estamos hablando de un objeto generado por una clase. Esta clase hereda de **junit.framework.TestCase**. Cada caso de prueba se implementa en un **método** aparte.

En el caso de los métodos de prueba veremos que siempre comienzan por *test*.

```
import junit.framework.*;  
  
public class CuentaTest extends TestCase {  
    ...  
}
```

Cuando realizamos un caso de prueba invocamos una serie de métodos que comprueban los resultados que se obtienen tras invocarlos.

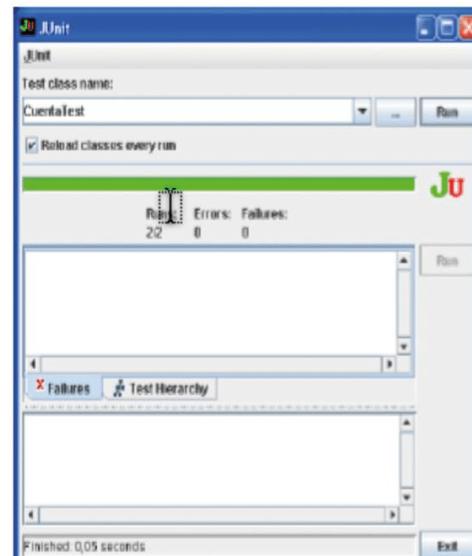
```
public void testCuentaNueva () {  
    Cuenta cuenta = new Cuenta();  
    assertEquals(cuenta.getSaldo(), 0.00);  
}  
  
public void testIngreso () {  
    Cuenta cuenta = new Cuenta();  
    cuenta.ingresar(100.00);  
    assertEquals(cuenta.getSaldo(), 100.00);  
}
```

Una vez que codifiquemos nuestros casos de prueba, la herramienta JUnit se encargará de validar los resultados y mostrarlos.



### Si el caso de prueba falla

Prueba de métodos JUnit.



### Funcionan correctamente

# Métodos JUnit

---

JUnit dispone de los métodos *Assert* para probar que los resultados sean los esperados por la ejecución de nuestro código. Estos métodos permiten especificar un error, el resultado esperado y el resultado real.

Estos métodos son:

- *assertArrayEquals()*
- *assertEquals()*
- *assertTrue()*
- *assertFalse()*
- *assertNull()*
- *assertNotNull()*
- *assertSame()*
- *assertNotSame()*
- *assertThat()*

Veamos algunos ejemplos de aplicación:

## assertArrayEquals()

```
package ar.com.ladooscurojava.model.test;

import org.junit.Assert;
import org.junit.Test;

/**
 * @author ciber
 */
public class PersonaTest {

    @Test
    public void testCompararStringArray() {

        String[] arrayEsperado = {"Juan", "María", "Jose"};
        String[] arrayPrueba = {"Juan", "María", "Jose"};

        Assert.assertArrayEquals(arrayEsperado, arrayPrueba);
    }

}
```

Con el método `assertArrayEquals()` comparamos el *array* de prueba con el *array* generado por nuestro código, determinando si el resultado es el esperado.

## assertEquals()

```
package ar.com.ladooscurojava.model.test;

import org.junit.Assert;
import org.junit.Test;

/**
 * @author ciber
 */
public class PersonaTest {

    @Test
    public void testComprarIgual() {
        Assert.assertEquals("dato1", "dato2");
    }

}
```

Con el método `assertEquals()` comprobamos si el dato resultante tiene el valor deseado.

## ***assertTrue() y assertFalse()***

```
package ar.com.ladooscurojava.model.test;

import org.junit.Assert;
import org.junit.Test;

/**
 * @author ciber
 */
public class VehiculoTest {

    @Test
    public void testComprarTrue() {
        Assert.assertTrue(true);
    }

}
```

Con los métodos *assertTrue()* y *assertFalse()* validamos si un resultado es verdadero o falso.

# Anotaciones JUnit

Para crear un método *test* dentro de una clase tenemos que añadir la anotación “*@test*”.

A continuación detallamos algunas anotaciones existentes:

- ***@RunWith***. Se le asigna una clase a la que se invocará en lugar del ejecutor por defecto.
- ***@Before***. Indicamos que el método se debe ejecutar antes de cada test (precede al método *setUp*).
- ***@After***. Indicamos que el siguiente método se debe ejecutar después de cada test.
- ***@Test***. Indicamos que es un método de test.

```
public class Empleado{  
  
    @Test  
    public void testSueldo(){  
        float resultadoReal = Empleado.calcularSueldo(2000);  
        float esperado = 2200;  
        assertEquals(resultadoReal, esperado, 0.01);  
    }  
  
}
```

Ejemplo de anotación *@Test*.

En algunos casos el resultado de la ejecución puede ser una excepción. Mediante la anotación `@Test` podemos indicar la excepción esperada.

```
@Test(expected=BRException.class)
public void testSueldo(){
    Empleado.sueldoCalc(2500);
}
```

Otra forma de realizar la prueba es utilizando el método `fail`, que nos indica un fallo en la prueba. De este modo, mediante un bloque `try-catch`, podemos capturar la excepción esperada si el método falla.

```
@Test
public void testSueldo(){
    try{
        Empleado.sueldoCalc(2500);
        fail('Se esperaba una excepcion')
    }catch(BRException e)
```

## Suites de pruebas

---

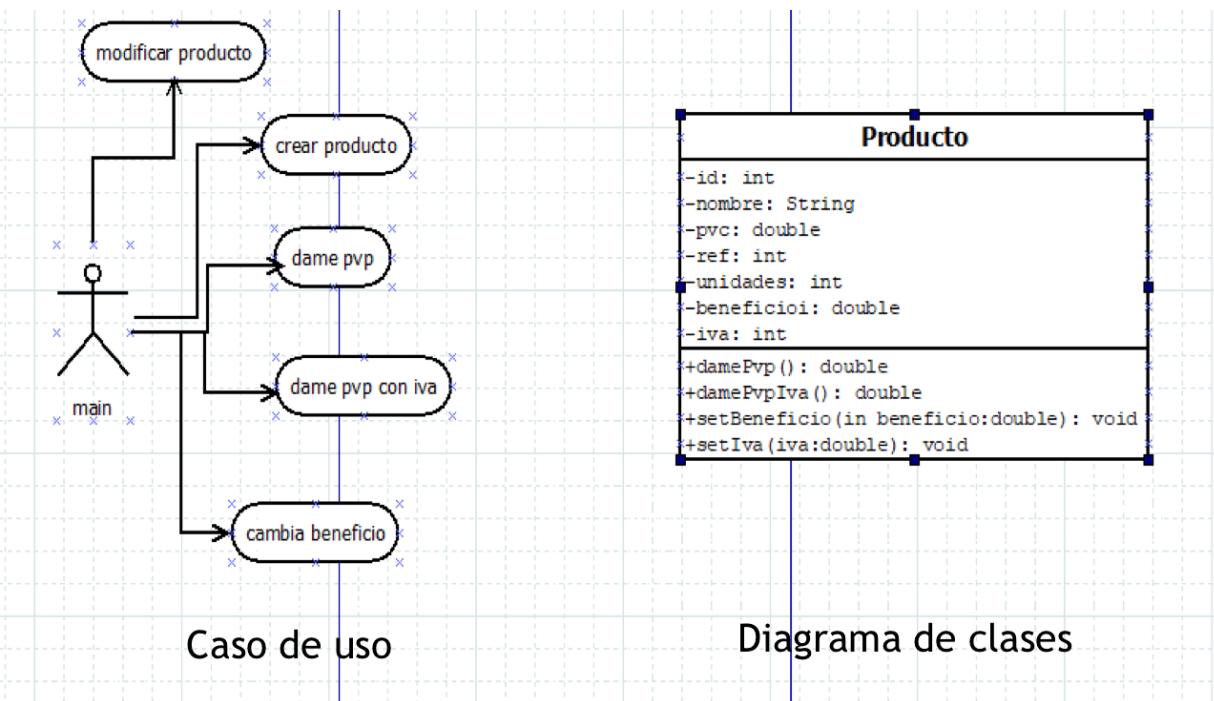
Cuando desarrollamos un programa la cantidad de métodos puede crecer considerablemente. Esto provoca que debamos agrupar diferentes casos de prueba para ejecutarlos a la vez.

Mediante las *suites* podemos agrupar métodos de prueba para ejecutarlos de forma conjunta.

```
@RunWith(Suite.class)
@SuiteClasses({Prueba1.class, Prueba2.class})
public class MiSuite{}
```

## Ejemplo de uso

Vamos a realizar un ejemplo de uso de JUnit sobre una aplicación muy sencilla que calcule el precio con IVA de un producto.



Diseño de nuestra aplicación.

```
public class Producto{  
  
    //ATRIBUTOS DE LA CLASE  
  
    private int id;  
    private String nombre;  
    private double pvc;  
    private int ref;  
    private int unidades;  
    private double beneficio;  
    private double iva;  
  
    public Producto() {  
        this.beneficio = 1.30;  
        this.iva = 1.21;  
    }  
  
    //CONSTRUCTORES  
    public Producto (int ref) {  
        this.ref = ref;  
        this.beneficio = 1.30;  
        this.iva = 1.21;  
    }  
  
    public Producto (int id, String nombre, double pvc, int ref, int unidades){  
  
        this.id = id;  
        this.nombre = nombre;  
        this.pvc = pvc;  
        this.ref = ref;  
        this.unidades = unidades;  
  
        this.beneficio = 1.30;  
        this.iva = 1.21;  
    }  
}
```

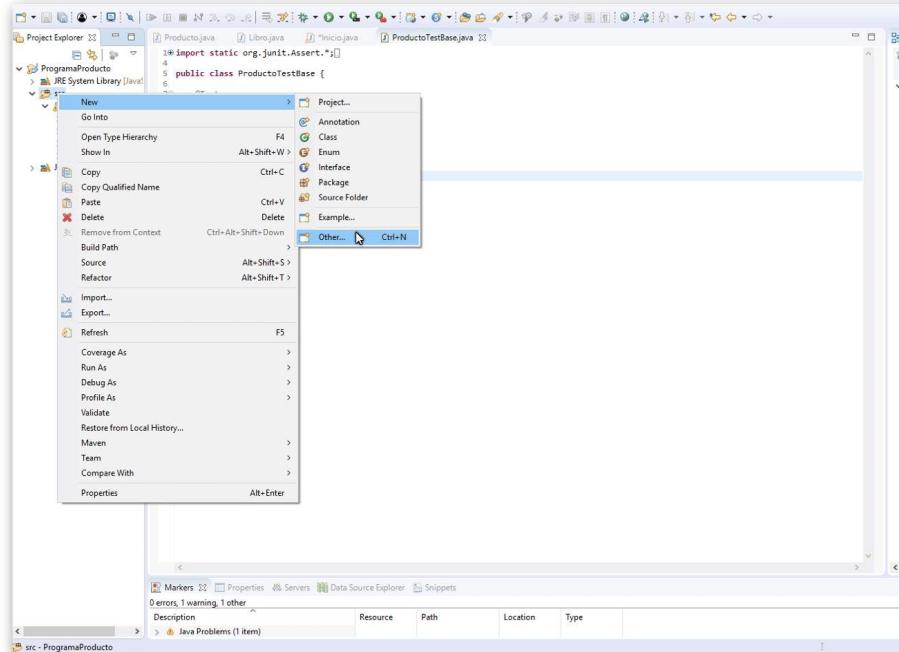
Declaración de atributos y constructores.

```
/*
*método que calcula un beneficio al coste del producto
*@return Valor total con el beneficio
*/
public double damePvp(){
    double pvp =  this.pvc * this.beneficio;
    return pvp;
}

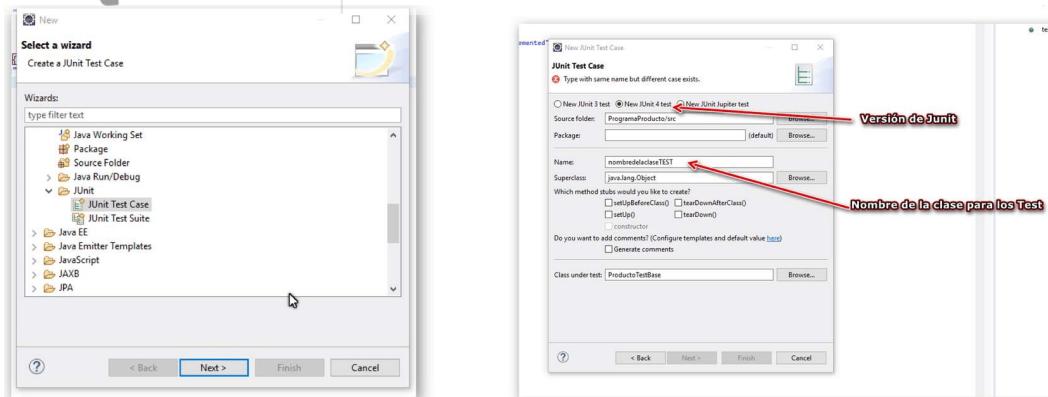
public double damePvpIva(){
    double res = this.damePvp() * this.iva;
    return res;
}
```

Un par de métodos para calcular el PVP y el precio con IVA.

Una vez tenemos nuestro pequeño programa, vamos a generar la clase de pruebas con JUnit.



Al ser la primera vez que lo utilizamos, Eclipse no nos mostrara el acceso en el menú inicial, por lo que le damos a "Other..." .



Seleccionamos la opción *JUnit Test Case* y en la ventana seleccionamos la versión de JUnit a utilizar y el nombre que le daremos a la clase de pruebas.

```

@Test
public void testDamePvp(){
    Producto prod = new Producto();

    //LÓGICA DE LA CLASE
    prod.setPvc(100.00);
    prod.setIva(1.21);
    prod.setBeneficio(1.3);
    double res = 130.00;

    //RESULTADO ESPERADO
    assertEquals(res, prod.damePvp(), 0.01);
}

@Test
public void testDamePvpIva(){
    Producto prod = new Producto();

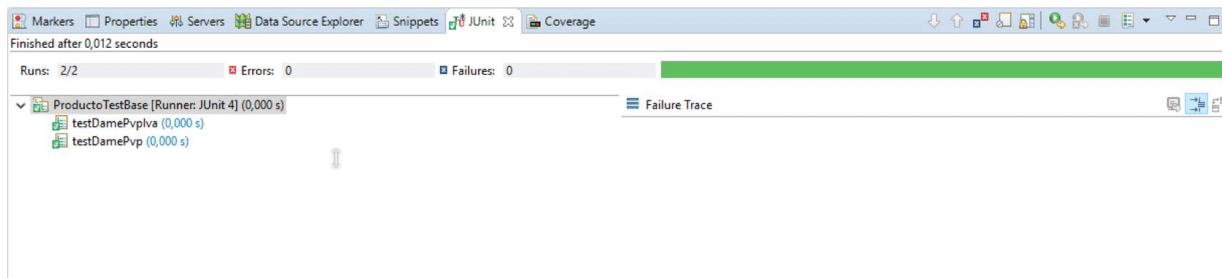
    //MÉTODO DE PRUEBA
    prod.setPvc(100.00);
    prod.setIva(1.21);
    prod.setBeneficio(1.3);

    //ESPERADO
    double res = 1.21*1.3*100.00;

    assertEquals (res, prod.damePvpIva(), 0.01);
}

```

Métodos de prueba, el primero compara el valor del precio con PVP y el segundo con el IVA añadido.



Resultado de la prueba.

---

# Pruebas parametrizadas

En algunas ocasiones no es suficiente comprobar el resultado con un solo dato, necesitando enviar una batería de datos para comprobar que el método funciona correctamente.

Con JUnit podemos parametrizar pruebas, enviando mediante una lista una serie de datos a comprobar.

```
@RunWith (Parameterized.class)
public class ProductoParametrosTest {

    private double pvc;
    private double esperado;

    public ProductoParametrosTest( double pvc, double esperado) {
        // TODO Auto-generated constructor stub
        Producto prod = new Producto();
        prod.setPvc(pvc);

        this.pvc = prod.damePvp();

        this.esperado = esperado;
    }

    @Parameters
    public static Collection<Object[]> data (){
        Object[][] = data = new Object [][] { {10.00,13.00} , {100.00, 130.00} };
        return Arrays.asList(data);
    }

    @Test
    public void test() {
        assertEquals(pvc, esperado, 0)
    }
}
```

Ejemplo de prueba parametrizada.

## Resumen

---

Has finalizado esta lección. En el siguiente documento descargable está todo el contenido que has visto.

En esta unidad hemos aprendido en qué consisten las pruebas unitarias que se suelen realizar con componentes de software asociados a bloques de código, clases o métodos.

La herramienta en la que generamos ese código suele ser el entorno de programación o IDE. El IDE es entonces el sitio correcto donde realizar pruebas para cada componente programado. NetBeans, Eclipse y otros IDE incorporan módulos externos para realizar estas pruebas, uno de los más famosos es JUnit.



**PROEDUCA**