

# Análisis de Caso

---

Testing Unitario

Ashley Rodriguez.

## **Situación Inicial.**

Una pequeña empresa de tecnología, SmartOrders, está desarrollando una aplicación de pedidos en línea. El equipo de desarrollo ha avanzado en la implementación de funcionalidades básicas, como la clase PedidoService (procesa pedidos), Producto (representa un artículo disponible en la tienda) y CarritoCompra (administra la lista de productos seleccionados por el usuario). Hasta ahora, las validaciones de los pedidos se realizan manualmente, generando tiempo de respuesta lento ante posibles errores. Para solucionar esto, el gerente de proyecto ha decidido formalizar el Testing Unitario con JUnit, reforzando la detección temprana de defectos y facilitando modificaciones posteriores (refactorizaciones, mejoras, etc.). Con ello, se busca:

- implementar aserciones claras para validar la lógica interna (sumas, descuentos, validaciones de stock).
- Utilizar el ciclo de vida de test (métodos con @BeforeEach, @AfterEach, etc.) para preparar y limpiar datos.
- Aplicar pruebas parametrizadas y testeo de excepciones para cubrir escenarios normales y casos borde.
- Emplear suites de test para organizar las pruebas de múltiples módulos de la aplicación.

## **1. Testing Unitario**

El testing unitario es una técnica de prueba de software que se enfoca en verificar el comportamiento de unidades individuales de código como métodos o funciones, de forma aislada. Esto garantiza que cada función haga lo que debe hacer, se detectan errores de forma temprana, producen menos bugs, y garantiza una refactorización más segura.

## **2. Relevancia del Testing Unitario para SmartOrders**

SmartOrders es una aplicación encargada de gestionar pedidos en línea, en la que funcionalidades como agregar productos a un carrito, calcular totales y procesar pedidos deben funcionar de forma constante.

La importancia del testing unitario para SmartOrders es que mejora la forma de proteger la evolución del sistema a través de pruebas unitarias. Al aplicar el testing unitario permite validar el comportamiento de cada clase individual (Producto, CarritoCompra, PedidoService) en aislamiento, detectando errores de forma temprana y facilitando futuras refactorizaciones.

### 3. Creación de Pruebas Básicas con Aserciones

\*pruebas unitarias simples para **agregarProducto** en **CarritoCompra** y **procesarPedido** en **PedidoService** (Empleando `assertEquals`, `assertTrue`, `assertFalse` )

```
J CarritoCompra.java 1 x
src > main > java > com > ejemplo > J CarritoCompra.java > Language Support for Java(TM) by Red Hat > CarritoCompra
1  package com.ejemplo;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class CarritoCompra {
7      private List<Producto> productos = new ArrayList<>();
8
9      public void agregarProducto(Producto producto) {
10         productos.add(producto);
11     }
12
13     public double calcularTotal() {
14         return productos.stream().mapToDouble(Producto::getPrecio).sum();
15     }
16
17     public int cantidadProductos() {
18         return productos.size();
19     }
20
21     public List<Producto> getProductos() {
22         return productos;
23     }
24 }
```

```
er  Ir  ...  ← →  testingunitario [Administrador]  🏠 ✓

J CarritoCompraTest.java ×

src > test > java > com > ejemplo > J CarritoCompraTest.java > ...

1  package com.ejemplo;
2  import static org.junit.jupiter.api.Assertions.assertEquals;
3  import static org.junit.jupiter.api.Assertions.assertTrue;
4  import org.junit.jupiter.api.Test;
5
6
7  public class CarritoCompraTest {
8
9      @Test
10     void testAgregarProducto() {
11         CarritoCompra carrito = new CarritoCompra();
12         Producto producto = new Producto(nombre:"Mouse", precio:25.0);
13         carrito.agregarProducto(producto);
14
15         assertEquals(expected:1, carrito.cantidadProductos());
16         assertTrue(carrito.getProductos().contains(producto));
17     }
18
19     @Test
20     void testCalcularTotal() {
21         CarritoCompra carrito = new CarritoCompra();
22         carrito.agregarProducto(new Producto(nombre:"Mouse", precio:25.0));
23         carrito.agregarProducto(new Producto(nombre:"Teclado", precio:45.0));
24
25         assertEquals(expected:70.0, carrito.calcularTotal(), delta:0.01);
26     }
27 }
28
29
30
```

```
J PedidoService.java ×

src > main > java > com > ejemplo > J PedidoService.java > Language Support for Java(TM) by Red Hat > 🏠 PedidoService

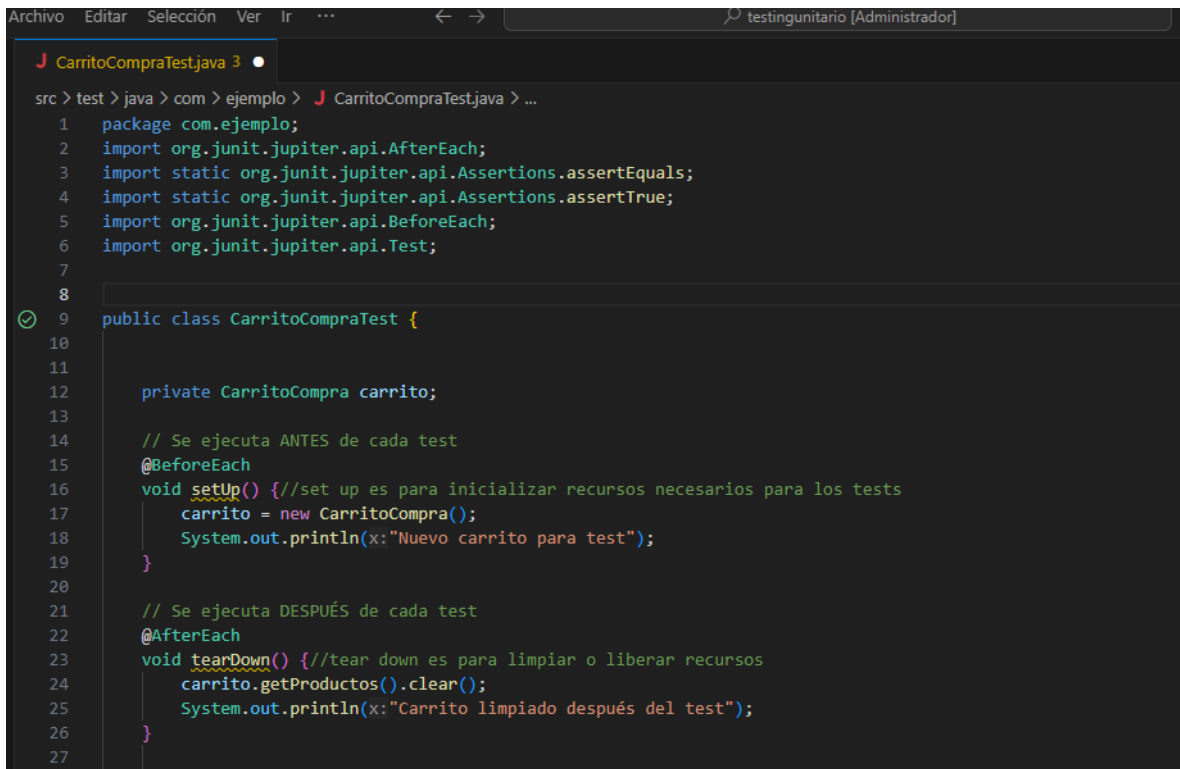
1  package com.ejemplo;
2
3  public class PedidoService {
4
5      public boolean verificarCupos(int cuposMaximos, int cuposOcupados) {
6          return cuposOcupados < cuposMaximos;
7      }
8  }
```

```
Producto.java X
src > main > java > com > ejemplo > Producto.java > Language Support for Java(TM) by Red Hat > Producto > Producto(String, double)
1 package com.ejemplo;
2
3 public class Producto {
4     private String nombre;
5     private double precio;
6
7     public Producto(String nombre, double precio) {
8         if (precio < 0) {
9             throw new IllegalArgumentException(s:"El precio no puede ser negativo");
10        }
11        this.nombre = nombre;
12        this.precio = precio;
13    }
14
15    public String getNombre() {
16        return nombre;
17    }
18
19    public double getPrecio() {
20        return precio;
21    }
22 }
```

```
Ver Ir ... testingunitario [Administrador]
ProductoTest.java 2 X
src > test > java > com > ejemplo > ProductoTest.java > Language Support for Java(TM) by Red Hat > ProductoTest > testPr
1 package com.ejemplo;
2
3 import org.junit.jupiter.api.Test;
4 import static org.junit.jupiter.api.Assertions.*;
5
6 public class ProductoTest {
7
8     @Test
9     void testCreacionProductoCorrecto() {
10        Producto producto = new Producto(nombre:"Mouse", precio:25.0);
11        assertEquals(expected:"Mouse", producto.getNombre());
12        assertEquals(expected:25.0, producto.getPrecio(), delta:0.01);
13    }
14
15    @Test
16    void testPrecioNegativoLanzaExcepcion() {
17        ....assertThrows(expectedType:IllegalArgumentException.class, ()->{
18            ....new Producto(nombre:"Monitor", -100.0);
19        });
20    }
21 }
```

## 2. Callbacks del Ciclo de Vida de Test

\*Implementa métodos `@BeforeEach` para inicializar un carrito nuevo o un set de productos antes de cada test, limpiando o reasignando datos en `@AfterEach` si es necesario.



```
Archivo  Editar  Selección  Ver  Ir  ...  testingunitario [Administrador]

J CarritoCompraTest.java 3 ●

src > test > java > com > ejemplo > J CarritoCompraTest.java > ...

1  package com.ejemplo;
2  import org.junit.jupiter.api.AfterEach;
3  import static org.junit.jupiter.api.Assertions.assertEquals;
4  import static org.junit.jupiter.api.Assertions.assertTrue;
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.Test;
7
8
9  public class CarritoCompraTest {
10
11
12     private CarritoCompra carrito;
13
14     // Se ejecuta ANTES de cada test
15     @BeforeEach
16     void setUp() { //set up es para inicializar recursos necesarios para los tests
17         carrito = new CarritoCompra();
18         System.out.println(x:"Nuevo carrito para test");
19     }
20
21     // Se ejecuta DESPUÉS de cada test
22     @AfterEach
23     void tearDown() { //tear down es para limpiar o liberar recursos
24         carrito.getProductos().clear();
25         System.out.println(x:"Carrito limpiado después del test");
26     }
27 }
```

El uso de `@BeforeEach` y `@AfterEach` asegura que cada prueba comience en un estado limpio y controlado. Por ejemplo, en `@BeforeEach` inicializamos un carrito nuevo antes de cada test, y en `@AfterEach` lo limpiamos para evitar que productos agregados en una prueba influyan en la siguiente.

## 3. Tests Parametrizados y Testeo de Excepciones

\*Diseña un test parametrizado que verifique distintos valores (por ejemplo, cantidades de producto válidas e inválidas) sin necesidad

de escribir varios métodos de prueba casi idénticos.

```
> test > java > com > ejemplo > J ProductoTest.java > ...
1 package com.ejemplo;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4 import static org.junit.jupiter.api.Assertions.assertNotNull;
5 import org.junit.jupiter.params.ParameterizedTest;
6 import org.junit.jupiter.params.provider.CsvSource;
7
8 public class ProductoTest {
9
10
11     @ParameterizedTest
12     @CsvSource({
13         "Mouse, 25.0",
14         "Teclado, 45.5",
15         "Monitor, 200.0"
16     })
17     void testCreacionProducto(String nombre, double precio) {
18         Producto producto = new Producto(nombre, precio);
19
20         assertNotNull(producto);
21         assertEquals(nombre, producto.getNombre());
22         assertEquals(precio, producto.getPrecio());
23     }
24 }
25
26
```

Un test parametrizado nos permite verificar múltiples valores de entrada y salida sin duplicar código. En este caso, probamos que un Producto se cree correctamente con diferentes nombres y precios, ahorrando tiempo y evitando escribir un método de prueba separado para cada caso.

```
public class ProductoTest {
    //...
    @Test
    void testPedidoConCantidadCeroLanzaExcepcion() {
        PedidoService service = new PedidoService();

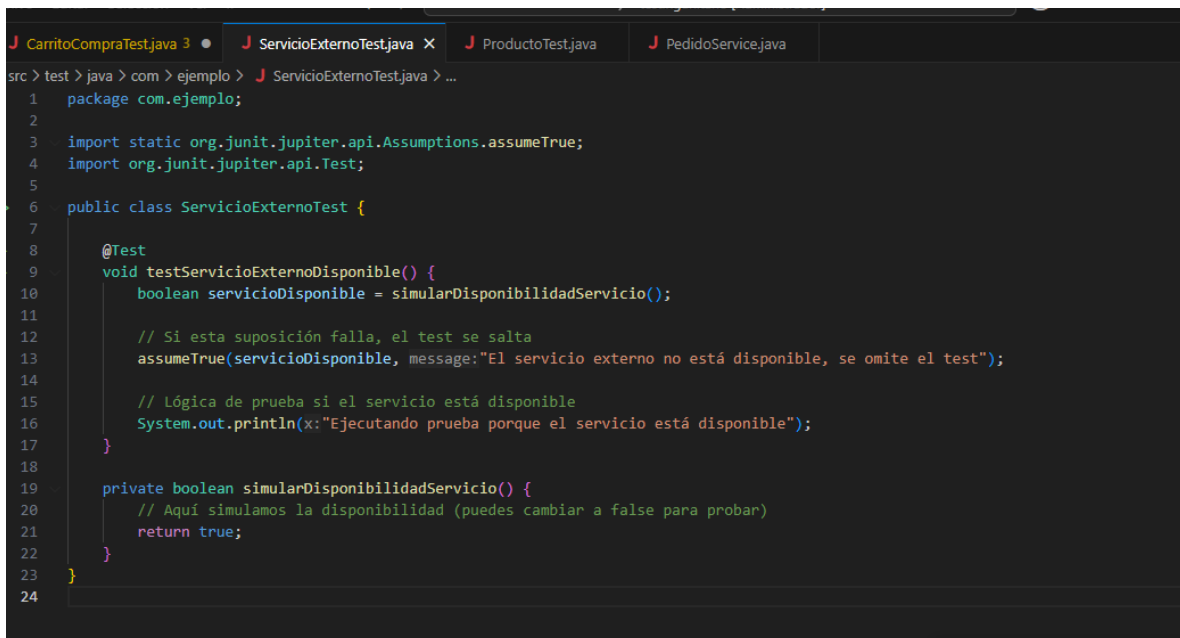
        IllegalArgumentException exception = assertThrows(
            IllegalArgumentException.class,
            () -> service.crearPedido(nombreProducto:"Mouse", cantidad:0)
        );

        assertEquals(expected:"La cantidad debe ser mayor a cero", exception.getMessage());
    }
    //...
    // @ParameterizedTest
    // ...
}
```

El test **testPedidoConCantidadCeroLanzaExcepcion** verifica que el sistema responde correctamente ante una entrada inválida, lanzando la excepción `IllegalArgumentException` con un mensaje específico.

#### **4. Uso de Suposiciones en JUnit**

\* Añade un test que utilice `assumeTrue` o `assumeFalse` para bifurcar la ejecución dependiendo de alguna condición (p.ej., la disponibilidad de un servicio externo simulado).



```
src > test > java > com > ejemplo > ServicioExternoTest.java > ...
1  package com.ejemplo;
2
3  import static org.junit.jupiter.api.Assumptions.assumeTrue;
4  import org.junit.jupiter.api.Test;
5
6  public class ServicioExternoTest {
7
8      @Test
9      void testServicioExternoDisponible() {
10         boolean servicioDisponible = simularDisponibilidadServicio();
11
12         // Si esta suposición falla, el test se salta
13         assumeTrue(servicioDisponible, message:"El servicio externo no está disponible, se omite el test");
14
15         // Lógica de prueba si el servicio está disponible
16         System.out.println(x:"Ejecutando prueba porque el servicio está disponible");
17     }
18
19     private boolean simularDisponibilidadServicio() {
20         // Aquí simulamos la disponibilidad (puedes cambiar a false para probar)
21         return true;
22     }
23 }
24
```

Las suposiciones (`assumeTrue` y `assumeFalse`) permiten que un test se ejecute solo si se cumple cierta condición. De esta forma, evitamos que un test falle por una causa ajena al código, manteniendo la integridad del reporte de pruebas.

#### **5. Organización en Suites**

\* Integra todas las pruebas creadas (`PedidoServiceTest`, `CarritoCompraTest`, `ProductoTest`, etc.) en una suite con `@RunWith(Suite.class)` o su equivalente en JUnit 5.



```

test > java > com > ejemplo > J TodasLasPruebasSuite.java > ...
package com.ejemplo;

import org.junit.platform.suite.api.SelectClasses;
import org.junit.platform.suite.api.Suite;

// Suite de pruebas para ejecutar todas juntas
@Suite
@SelectClasses({
    CarritoCompraTest.class,
    ProductoTest.class,
    PedidoServiceTest.class,
    ServicioExternoTest.class
})
public class TodasLasPruebasSuite {
}

```

Las suites de pruebas permiten ejecutar varios casos de prueba relacionados en un solo comando, facilitando la integración continua y reduciendo el tiempo de ejecución manual.

## **Reflexión Final**

Para asegurar una cobertura amplia y confiable, se aplicaron diversas técnicas modernas de pruebas unitarias con **JUnit 5**, que permiten verificar tanto los caminos esperados como los escenarios de error.

### **Callbacks (@BeforeEach, @AfterEach)**

Se utilizaron para preparar y limpiar el entorno de prueba antes y después de cada test

### **Pruebas parametrizadas (@ParameterizedTest)**

Fueron aplicadas en la clase ProductoTest para probar múltiples precios válidos sin duplicar métodos.

### **Testeo de excepciones (assertThrows)**

Fue fundamental para validar casos bordes como intentar crear un producto con precio negativo. Esta técnica asegura que el sistema reaccione correctamente ante datos inválidos, protegiendo la integridad del negocio.

Cada clase de prueba está orientada a garantizar la confiabilidad de un componente clave del sistema:

- **CarritoCompraTest** valida que los productos se agregan correctamente, que el total se calcula bien y que no hay datos residuales entre pruebas.
- **PedidoServiceTest** asegura que no se puede procesar un pedido vacío.
- **ProductoTest y ProductoExcepcionTest** verifican que los productos no puedan tener precios negativos.
- **PedidoServiceAssumeTest** demuestra cómo el sistema se comporta ante servicios externos disponibles o no.
- **AllTestsSuite** agrupa todas las pruebas en una sola suite ejecutable.

En conjunto, estas pruebas aumentan la estabilidad, la calidad y la mantenibilidad de SmartOrders.