ISFDyT N°70

Tecnicatura Superior en Análisis de Sistemas

Ingeniería de Software II

- Trabajo Práctico 1 – Introducción a JUnit
- Eguia Juan Manuel
- 3° año
- **Nombre de la profesora:** Marina Caseres
- **Fecha de entrega:** 12/09/2025
- Enlace al repositorio GitHub: https://github.com/juaneguia/TP1_JUnit_EguiaJuanManuel
- juanma.eguia@gmail.com

TRABAJO PRACTICO 1 – INTRODUCCIÓN A JUNIT

El programa seleccionado corresponde a una tienda informática, cuyo objetivo es gestionar distintos productos y calcular sus precios finales.

La estructura del sistema se basa en una clase padre ***Producto***, que define los atributos y comportamientos generales comunes a todos los artículos de la tienda. A partir de esta clase se derivan dos clases hijas:

Computadora: representa computadoras, incorporando las particularidades necesarias para el cálculo de su precio final.

Impresora: representa impresoras, incluyendo reglas específicas que influyen en el costo total.

Además, se implementa la clase ***Cajero***, encargada de calcular el monto final de la compra considerando la cantidad de productos adquiridos.

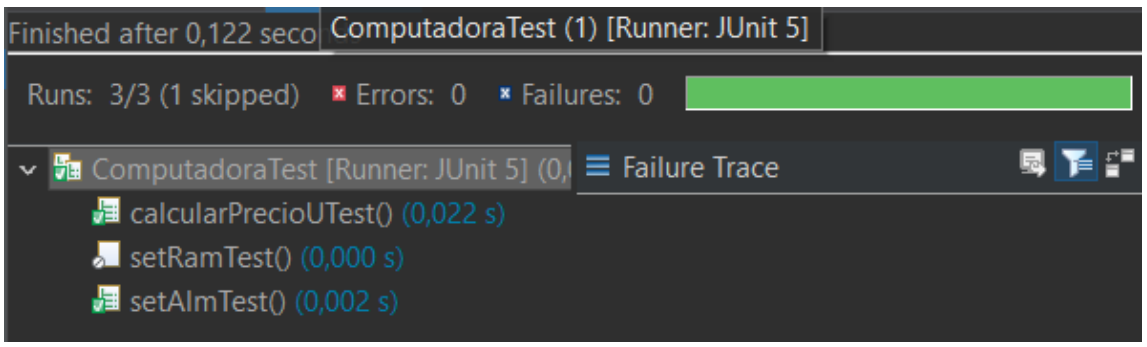
Finalmente, en la clase ***Main*** se instancian objetos con diferentes características, y se utilizan las funcionalidades del sistema para obtener el precio de cada producto y el total de la compra.

CLASES TESTEADAS

CLASE	MÉTEDO	DESCRIPCIÓN
Computadora	calcularPrecioU() setRam() setAlmacenamiento()	Calcula el precio final de la unidad Modifica el valor del atributo Ram Modifica el valor del atributo Ram
Impresora	calcularPrecioU() isColor() setStock()	Calcula el precio final de la unidad Verifica si el color es verdadero o falso Verifica si el parámetro es válido y lo modifica
Cajero	calcularPrecioFinal() setCantidad() calcularPrecioFinal()	Calcula el precio final de todos los productos Verifica si el parámetro es válido y lo modifica Calcula el precio final (con excepción)

CLASE COMPUTADORA TEST

```java
class ComputadoraTest {

    @Test
    public void calcularPrecioUTest() {
        Computadora c1 = new Computadora(1000, "hp", 10, "dsmsmd", 16, 512, false);

        double resultado = c1.calcularPrecioU();
        assertNotEquals(19519, resultado);
    }

    @Test
    @Disabled ("Test no realizado")
    public void setRamTest() {
        Computadora c2 = new Computadora(1000, "hp", 10, "dsmsmd", 16, 512, false);

        int nuevaRam = c2.setRam(32);
        assertEquals(32, nuevaRam);
    }

    @Test
    public void setAlmTest() {
        Computadora c3 = new Computadora(1000, "hp", 10, "dsmsmd", 16, 512, false);

        assertThrows(NullPointerException.class, () -> {
            c3.setAlmacenamiento((Integer) null);
        });
    }
```



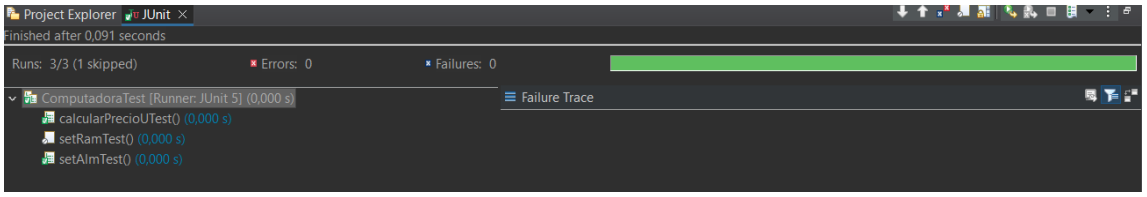
calcularPrecioUTest(): En este caso el método debía calcular el precio de la computadora y el valor de la misma era de 1000. Por medio de un assertNotEquals le indicamos que el resultado NO debía ser 19519, lo cual se verifico correctamente con el test.

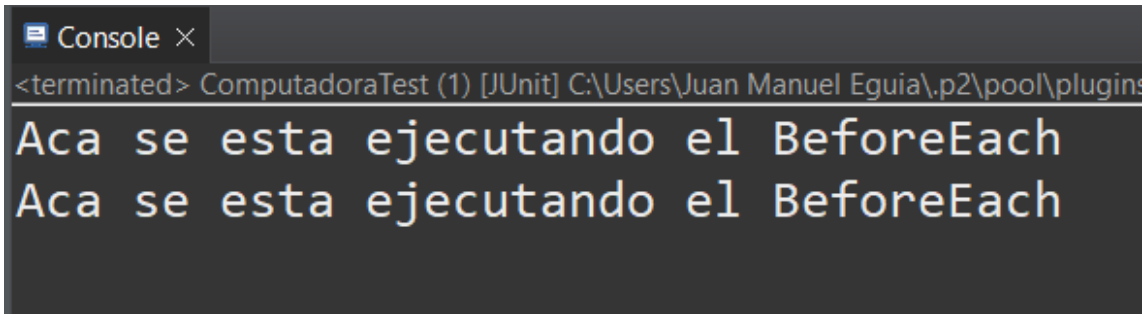
setRamTest(): Inicialmente, el método modificaba el valor de la RAM del objeto ya creado y por medio de un assertEquals comparamos el valor esperado con el ingresado, en este caso en ambos el valor era de 32, por lo que el testeo validaba correctamente estos datos. Aunque este testeo quedo deshabilitado.

setAlmTest(): El método modificaba el valor del almacenamiento del objeto creado, el cual su valor no puede ser nulo, entonces por medio de un assertThrows introducimos un Null, en este caso debería saltar la excepción y por medio del test se llegó a ese resultado esperado.

CLASE COMPUTADORA TEST (Modificada para agregar *@BeforeAll @BeforeEach)*

```java
class ComputadoraTest {

    static Computadora c1;
    @BeforeAll
    static void inicializar() {
     c1 = new Computadora(1000, "hp", 10, "dsmsmd", 16, 512, false);
    }
    @BeforeEach
    void mensaje() {
        System.out.println("Aca se esta ejecutando el BeforeEach");
    }

    @Test
    public void calcularPrecioUTest() {

        double resultado = c1.calcularPrecioU();
        assertNotEquals(19519, resultado);
    }

    @Test
    @Disabled ("Test no realizado")
    public void setRamTest() {

        int nuevaRam = c1.setRam(32);
        assertEquals(32, nuevaRam);
    }
```



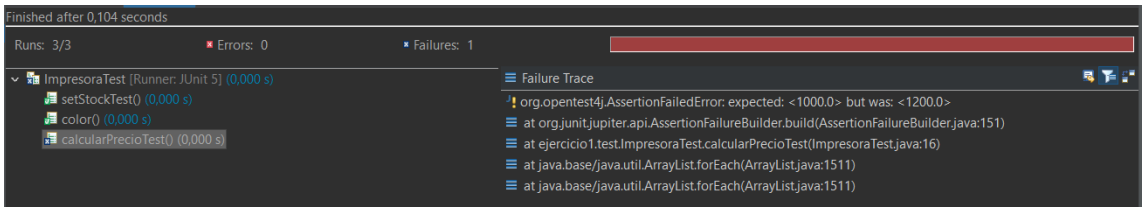


Se agrega *@BeforeAll para inicializar una variable estática (en este caso instanciando un objeto de Computadora) que se ejecutara antes de cada método. Posteriormente, el testeo valido nuevamente los resultados sin existir errores.*

Se agrega @BeforeEach el cual en este caso mostrara un mensaje antes de ejecutar cada método mostrándolo por consola. Se presento por consola solo dos veces, ya que uno de los métodos estaba deshabilitado, si no lo hubiese estado, el mensaje se hubiese mostrado tres veces, ya que eran tres métodos.

CLASE IMPRESORA TEST

```java
class ImpresoraTest {

    @Test
    public void calcularPrecioTest() {
        Impresora i1 = new Impresora(1000, "hp", 20, 10, true);

        double resultado = i1.calcularPrecioU();
        assertEquals(1000, resultado);
    }

    @Test
    public void color() {
        Impresora i2 = new Impresora(1000, "hp", 20, 10, true);
        assertTrue(i2.isColor());
    }

    @Test
    public void setStockTest() {
        Impresora i2 = new Impresora(1000, "hp", 0, 10, true);

        assertThrows(IllegalArgumentException.class, () -> {
            i2.setStock(0);
        });}
}
```



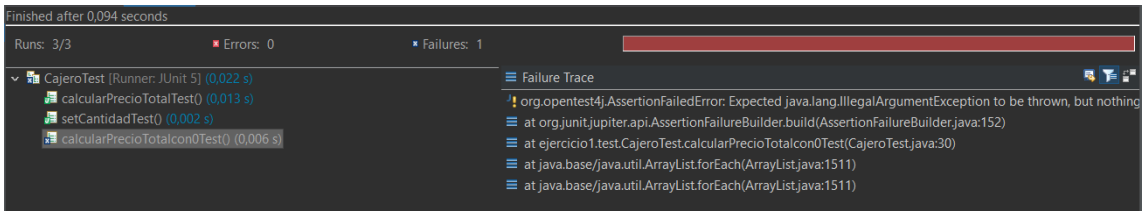
calcularPrecioTest(): La impresora tenía un valor de 1000, pero se le debía agregar 200 más por tener la condición true en color, dando un valor total de 1200. Por medio de un assertEquals le indicamos que el resultado debía ser 1000, por lo que el resultado del test no fue el esperado indicando un error.

color(): La impresora tiene un booleano si imprime o no a color. El método verifica la condición y en este caso con un assertTrue el testeo verifico que la condición era verdadera.

setStock(): El método modifica la cantidad de stock, pero no está permitido darle el valor de 0. Se comprobó con un assertThrows asignando el valor de 0, que lanza la excepción porque lo que es testeo es correcto.

CLASE CAJERO TEST

```java
class CajeroTest {

    @Test
    public void calcularPrecioTotalTest() {
        Producto p1 = new Impresora(10000, "Epson", 200, 10, false);
        Cajero c1 = new Cajero("Maria", p1, 10);
        double resultado = c1.calcularPrecioFinal();
        assertEquals(100000, resultado);
    }

    @Test
    public void setCantidadTest() {
        Producto p1 = new Impresora(10000, "Epson", 200, 10, false);
        Cajero c1 = new Cajero("Maria", p1, 10);
        assertThrows(IllegalArgumentException.class, () -> {
        c1.setCantidad(0);
        });}

    @Test
    public void calcularPrecioTotalcon0Test() {
        Producto p1 = new Impresora(10000, "Epson", 200, 10, false);
        Cajero c1 = new Cajero("Maria", p1, 1);
        assertThrows(IllegalArgumentException.class, () -> {
            c1.calcularPrecioFinal();
        });
    }
```



calcularPrecioTotalTest(): Una computadora cuesta 10000 y la cantidad son 10, por lo que el valor final y esperado es 100000. Se verificó que el método calcula correctamente el total de una compra sumando los precios unitarios de los productos por su cantidad. El resultado coincidió con el esperado mediante assertEquals.

setCantidadTest(): El método modifica la cantidad de productos, el cual NO puede ser 0. Se probó que acepte valores positivos y lance excepción al intentar cargar valores no válidos (0 o negativos). El test con assertThrows pasó exitosamente.

calcularPrecioTotalcon0Test() El método calcula el precio final en este caso con valores válidos. Se pretende con assertThrows lanzar la excepción multiplicando por 0 lo cual no puedo hacer, y en el resultado del testeo nos devolvió error.

REFLEXIÓN FINAL

El uso de JUnit y las pruebas unitarias resulta fundamental en el desarrollo de software, ya que permiten comprobar que cada método cumpla con su funcionalidad prevista. Gracias a estas pruebas, se pueden detectar errores en etapas tempranas del desarrollo, evitando que se propaguen a otras partes del sistema.

Además, trabajar con pruebas unitarias genera mayor confianza en el código, facilita el mantenimiento y sirve como documentación viva sobre el comportamiento esperado de cada clase y método. En nuestro caso, las pruebas permitieron verificar no solo los cálculos correctos, sino también el manejo adecuado de valores inválidos mediante excepciones.