16-3-2023

Tema 4:

Diseño y realización de pruebas.

Gerardo Pérez Macareno – 1ºDAM
GRUPO STUDIUM

Índice:

Introducción:

En este tema aprenderemos y desarrollaremos un tema tan importante como útil como son las pruebas de caja negra y caja blanca.

Estas son importantes en el proceso de prueba de software, ya que verifica que este funciona correctamente y cumple con los requisitos y especificaciones que deseamos.

Las pruebas de caja negra se centran en probar la funcionalidad externa del software, es decir, su comportamiento desde la perspectiva del usuario o de otros sistemas con los que interactúa, sin considerar su implementación interna. Se pueden usar para probar varios escenarios y situaciones que pueden ocurrir cuando se usa el software y verificar que el resultado esperado se está produciendo correctamente.

Las pruebas de caja blanca se enfocan en verificar la implementación interna del software, es decir, su lógica y estructura. Estas pruebas examinan el código fuente y las estructuras de datos utilizadas en el software, se utilizan para verificar que el software cumple con los estándares de codificación y las mejores prácticas, y detectan posibles errores en el código.

Estas pruebas también ayudan a detectar problemas al principio del desarrollo y ayudan a identificar y corregir errores.

Para aprender más sobre el tema haremos dos ejercicios ilustrativos uno de prueba de caja blanca y otro de caja negra.

Prueba de Caja Blanca:

Se pide:

• Crear grafo de recubrimiento indicando gráficamente los posibles caminos que puede tomar la ejecución del código. Calcula la complejidad ciclomática y di qué nos indica.

• Identifica los diferentes caminos. Crea tantos casos de prueba como caminos haya.

• Codificar las pruebas con JUnit que permitan validar el correcto funcionamiento de los métodos.

• Adjuntar el resultado de la ejecución de las pruebas.

```
1  public class Examen
2  {
3  public static String mensaje (int edad)
4  {
5  switch (edad)
6  {
7  case 18:
8  return "Ya eres mayor de edad";
9  case 67:
10 return "Ya puedes jubilarte";
11 default:
12 return "No pasa nada";
13 }
14 }
15 public static boolean validar (String dni)
16 {
17 if (dni.length() >= 9)
18 {
19 return true;
20 }
21 else
22 {
23 return false;
24 }}
25 }
26
```

1º Método:

```java
public static String mensaje (int edad)
{
switch (edad)
{
case 18:
return "Ya eres mayor de edad";
case 67:
return "Ya puedes jubilarte";
default:
return "No pasa nada";
}
}
```

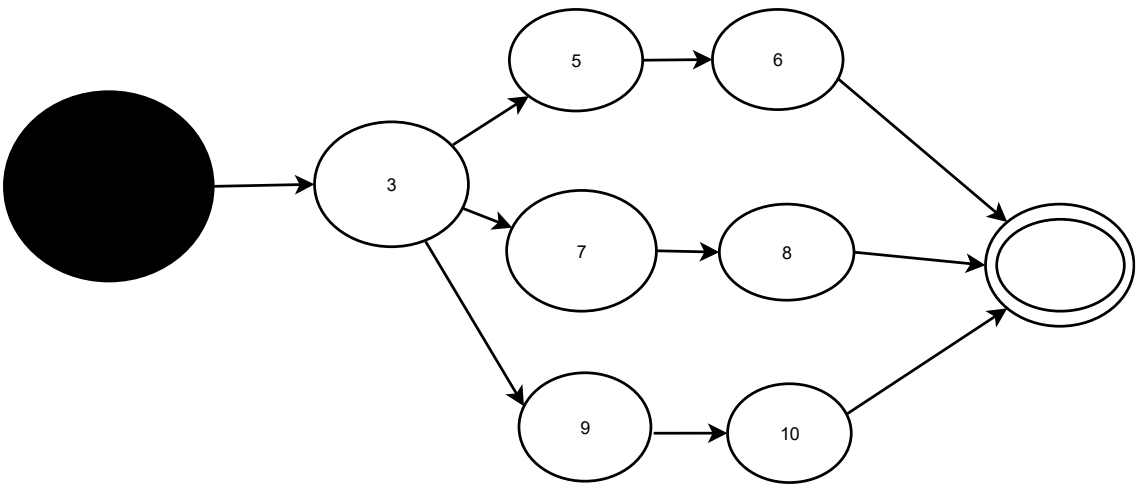
Este método denominado "mensaje" toma un argumento entero "edad" y devuelve una cadena de texto en función de ese valor de edad.

El método utiliza la estructura de switch para comparar el valor de "edad" con dos casos específicos: 18 y 67. Si "edad" es igual a 18, se devuelve el mensaje "Ya eres mayor de edad". Si "edad" es igual a 67, se devuelve el mensaje "Ya puedes jubilarte". Si el valor de "edad" no coincide con ninguno de estos casos, se devuelve el mensaje "No pasa nada" que es un mensaje por defecto.

Grafo de recubrimiento:

Nos referiremos a una técnica utilizada para diseñar casos de prueba para probar la funcionalidad de un programa.

En esta técnica, el programa se representa como un grafo, donde los nodos representan las distintas partes del programa, como, por ejemplo, las funciones, las sentencias de control, las entradas de datos, las salidas de datos, etc. Las flechas del grafo representan las interacciones entre estas partes del programa, como las llamadas de funciones, las transferencias de datos, las condiciones de control, etc.



Calcular Complejidad Ciclomática:

Complejidad ciclomática = número de ramas - número de nodos + 2

En el caso de este programa el resultado será: 10-9+2 = 3.

Mide la cantidad de decisiones que se deben tomar en un programa para cubrir todos los posibles caminos a través del programa. Cada vez que se toma una decisión, se crea un nuevo camino independiente, lo que aumenta la complejidad ciclomática.

Nos quiere decir que el número de caminos independientes de este diagrama de flujo será de 3.

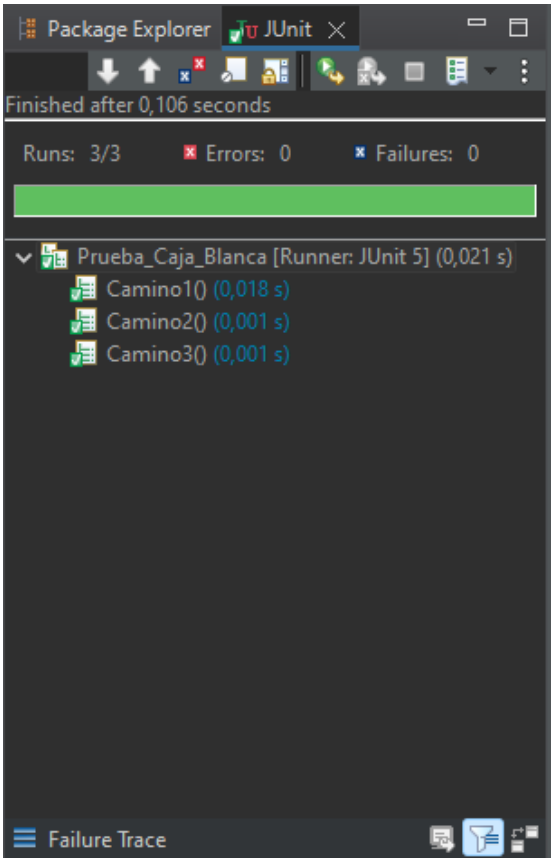
Identifica los diferentes caminos. Crea tantos casos de prueba como caminos haya:

Caminos	
1	Inicio,3,5,6,fin
2	Inicio,3,7,8,fin
3	Inicio,3,9,10,fin

Camino	Casos de Prueba	Resultado Esperado
1	mensaje(18)	"Ya eres mayor de edad"
2	mensaje(67)	"Ya puedes jubilarte"
3	mensaje(25)	"No pasa nada"

Codificar las pruebas con JUnit que permitan validar el correcto funcionamiento de los métodos.

```java
package es.studium.Tarea4;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;


class Prueba_Caja_Blanca
{

    @Test
    void Camino1()
    {
        String resultadoEsperado = "Ya eres mayor de edad";
        String resultadoReal = Examen.mensaje(18);
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino2()
    {
        String resultadoEsperado = "Ya puedes jubilarte";
        String resultadoReal = Examen.mensaje(67);
        assertEquals(resultadoEsperado, resultadoReal);

    }

    @Test
    void Camino3()
    {
        String resultadoEsperado = "No pasa nada";
        String resultadoReal = Examen.mensaje(25);
        assertEquals(resultadoEsperado, resultadoReal);
    }

}
```



2º Método:

```
1  public static boolean validar (String dni)
2  {
3  if (dni.length() >= 9)
4  {
5  return true;
6  }
7  else
8  {
9  return false;
10 }
11 }
12
```

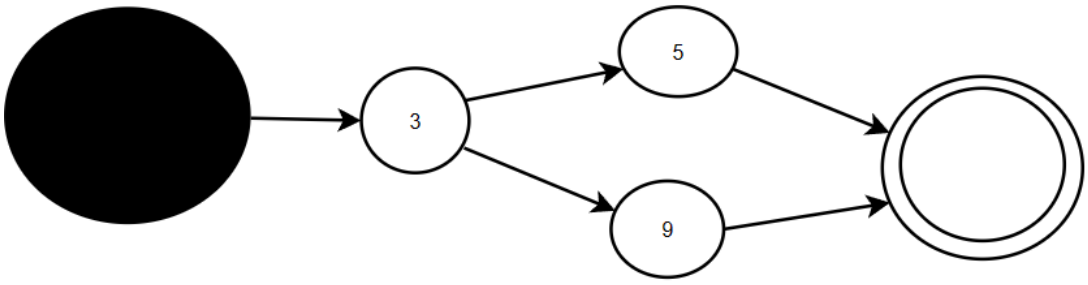
Este método denominado "validar" toma una cadena de caracteres (String) como entrada y devuelve un valor booleano como resultado.

La función verifica si la longitud de la cadena "dni" es mayor o igual a 9 caracteres. Si la longitud es mayor o igual a 9, la función devuelve "true", lo que indica que el "dni" es válido. Si la longitud es menor a 9, la función devuelve "false", lo que indica que el "dni" no es válido.

Grafo de recubrimiento:

Nos referiremos a una técnica utilizada para diseñar casos de prueba para probar la funcionalidad de un programa.

En esta técnica, el programa se representa como un grafo, donde los nodos representan las distintas partes del programa, como, por ejemplo, las funciones, las sentencias de control, las entradas de datos, las salidas de datos, etc. Las flechas del grafo representan las interacciones entre estas partes del programa, como las llamadas de funciones, las transferencias de datos, las condiciones de control, etc.



Calcular Complejidad Ciclomática:

Complejidad ciclomática = número de ramas - número de nodos + 2

En el caso de este programa el resultado será: 5-5+2 = 2.

Mide la cantidad de decisiones que se deben tomar en un programa para cubrir todos los posibles caminos a través del programa. Cada vez que se toma una decisión, se crea un nuevo camino independiente, lo que aumenta la complejidad ciclomática.

Nos quiere decir que el número de caminos independientes de este diagrama de flujo será de 2.

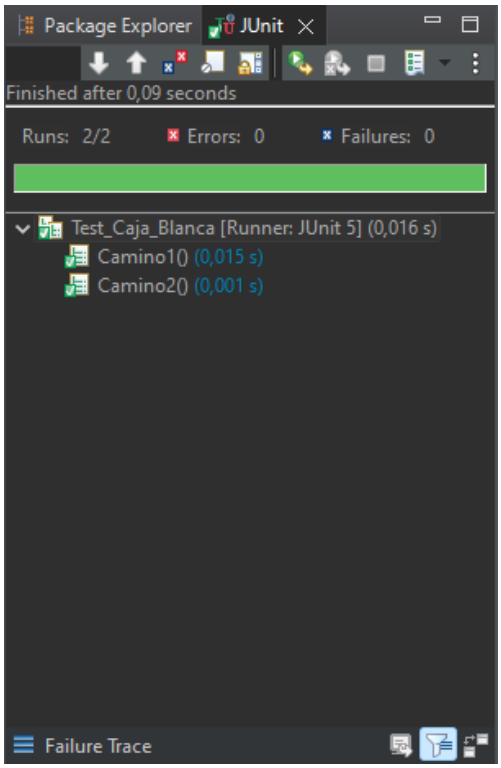
Identifica los diferentes caminos. Crea tantos casos de prueba como caminos haya:

Caminos	
1	Inicio,3,5,fin
2	Inicio,3,9,fin

Camino	Casos de Prueba	Resultado Esperado
1	validar("30250334K")	true
2	validar("3250334K")	false

Codificar las pruebas con JUnit que permitan validar el correcto funcionamiento de los métodos.

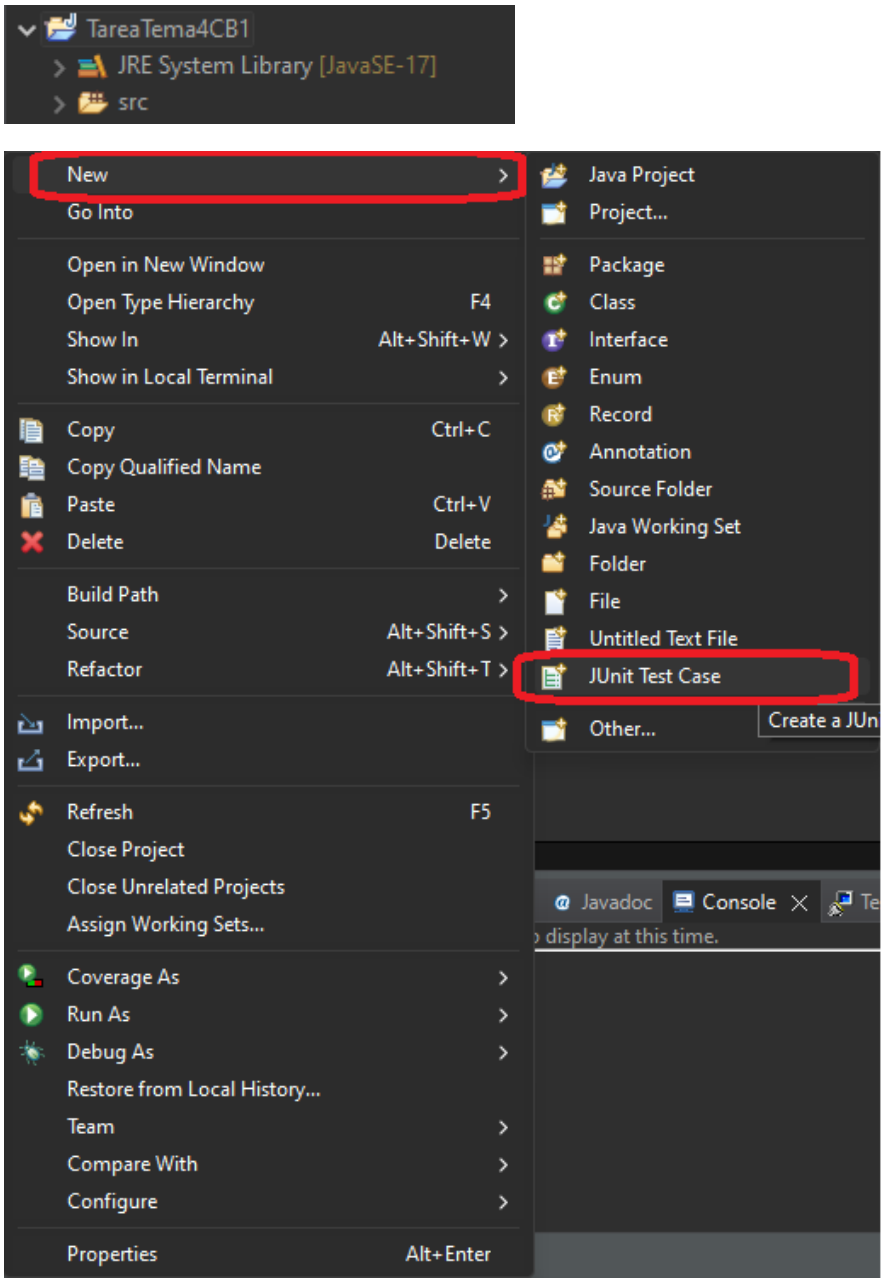
```java
package es.studium.tarea4;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;



class Test_Caja_Blanca
{

    @Test
    void Camino1()
    {
        boolean resultadoEsperado = true;
        boolean resultadoReal = Examen.validar("30250334K");
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino2()
    {
        boolean resultadoEsperado = false;
        boolean resultadoReal = Examen.validar("3250334K");
        assertEquals(resultadoEsperado, resultadoReal);

    }
}

```

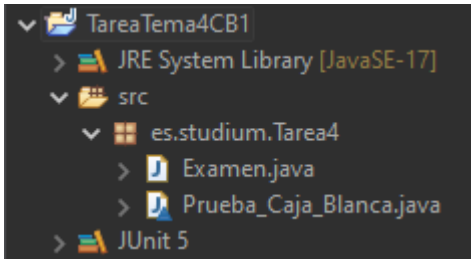


JUNIT, que es y cómo desarrollarlo:

Junit es un marco de pruebas unitarias, que nos permite escribir y ejecutar pruebas automatizadas de manera efectiva y eficiente. Las pruebas unitarias son pruebas que se enfocan en probar una sola unidad de código, como una clase o un método, de forma aislada y repetible.

Para crearlo, haremos clic derecho en el proyecto que deseemos y en el despegable, iremos a “New” y seleccionaremos “JUnit Test Case”.





Si el proceso se ha realizado con éxito veremos en nuestro proyecto la librería Junit y un nuevo archivo que será la prueba en sí con un triángulo azul identificativo.

Para crearlo iremos desarrollando paso a paso que es lo que necesitamos:

1º. Anotaciones: Junit utiliza anotaciones como @Test para identificar los métodos que contienen pruebas.

2º. Identificamos que es lo que nos quiere devolver el programa en este ejemplo de programa es un "boolean", luego creamos una variable denominada "resultadoEsperado" en la que guardaremos el valor que habíamos encontrado en los casos de prueba.

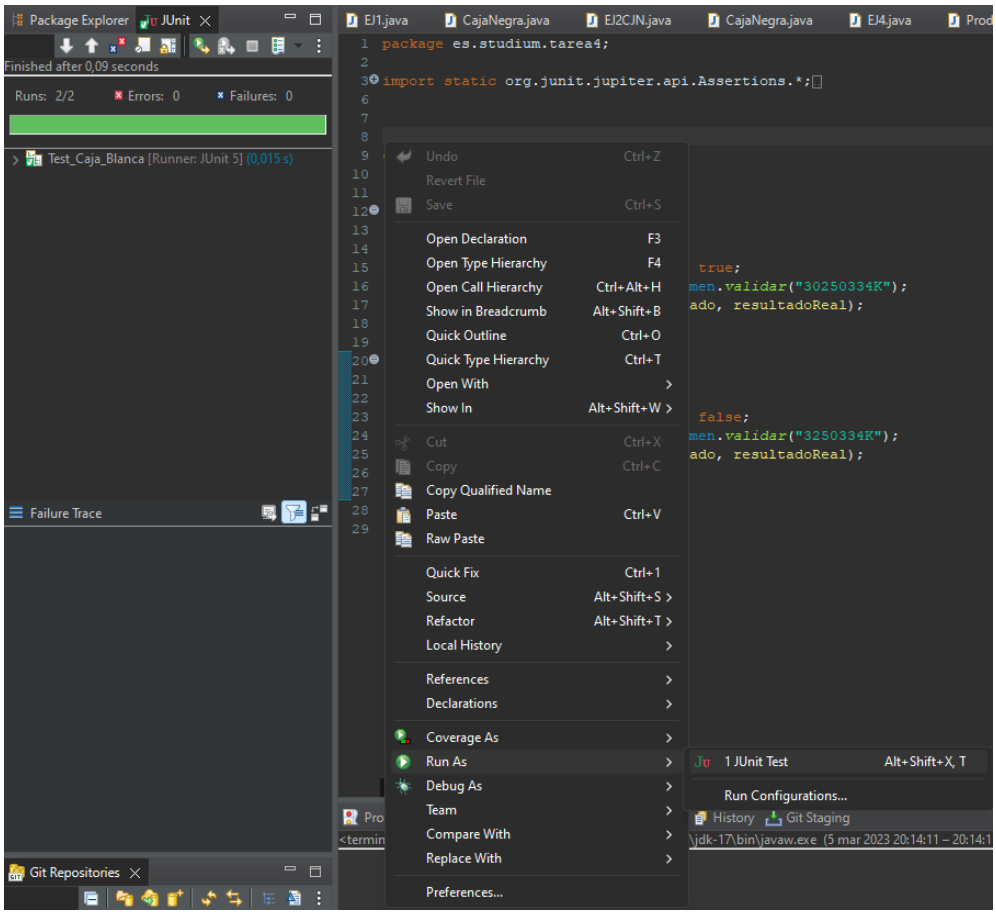
3º. Identificamos que es lo que nos quiere devolver el programa en este ejemplo de programa es un "boolean", luego creamos una variable denominada "resultadoReal" en la que guardaremos el valor que habíamos encontrado en los casos de prueba, deberemos tener cuidado ya que no le podrá faltar ningún carácter a esta cadena. Aquí llamaremos al método y a su función.

4º. Assertions: Junit proporciona una variedad de métodos de aserción, como assertEquals, para verificar que los resultados esperados se cumplan ya que comparará ambos y verán si son iguales.

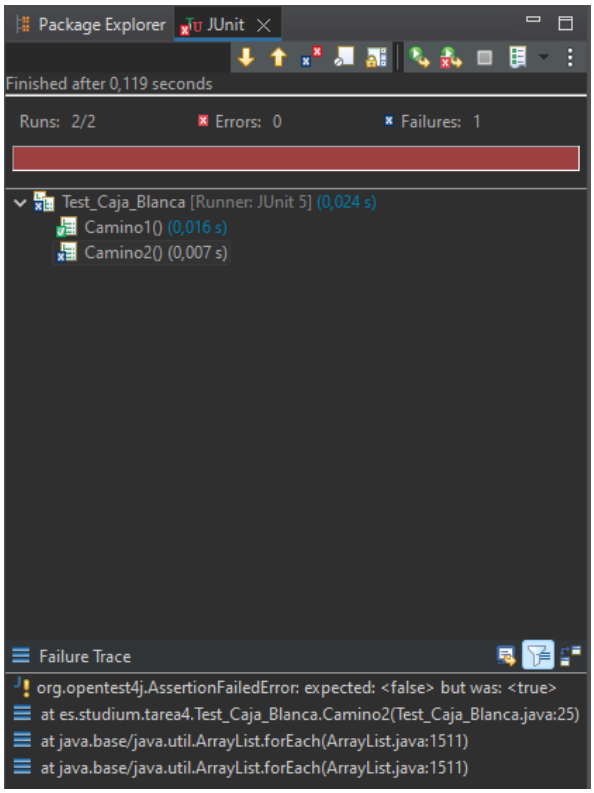
```
1  package es.studium.tarea4;
2
3  import static org.junit.jupiter.api.Assertions.*;
6
7
8
9  class Test_Caja_Blanca
10 {
11
12     @Test
13     void Caminol()
14     {
15         boolean resultadoEsperado = true;
16         boolean resultadoReal = Examen.validar("30250334K");
17         assertEquals(resultadoEsperado, resultadoReal);
18     }
19
20     @Test
21     void Camino2()
22     {
23         boolean resultadoEsperado = false;
24         boolean resultadoReal = Examen.validar("3250334K");
25         assertEquals(resultadoEsperado, resultadoReal);
26
27     }
28 }
```

Para validar nuestra prueba haremos clic derecho en el programa, “Run As”, “JUnit Test”.

Si el proceso se ha realizado con éxito veremos a nuestra izquierda la barra verde de confirmación.



En caso contrario Eclipse nos avisará con una barra de color rojo. Nos indicará también donde está el fallo y de que tipo es.



Prueba de Caja Negra:

Diseñamos la clase Producto, que implementa el funcionamiento básico del producto de un supermercado, donde tenemos tres operaciones: establecer un precio, aplicar un descuento y cambiar el IVA que se aplica al producto (por defecto es 21, pero podría ser el 10 o el 4).

Si se ejecuta cualquier método con valores incorrectos, no cambian los atributos del objeto.

Se pide:

1. Obtener las clases de equivalencia de las entradas y salidas de los métodos aplicarDescuento(), setPrecio() y setIVA() de la clase Producto.
2. A partir de las clases de equivalencia obtener los casos de prueba de caja negra.
3. Codificar las pruebas con JUnit para validar el correcto funcionamiento de la clase.

```java
public class Producto
{
private double precio;
private int iva;
public Producto()
{
precio = 0.0;
iva = 21;
}
public void aplicarDescuento (int porcentaje)
{
}
public void setIVA (int iva)
{
}public void setPrecio(double precio)
{
}
public int getIVA ()
{
}
public double getPrecio ()
{
}
}
```

1º. Obtener las clases de equivalencia de las entradas y salidas de los métodos aplicarDescuento(), setPrecio() y setIVA() de la clase Producto.

1º Clases de equivalencia:

Se evalúa para las entradas tanto para los:

-Valores Válidos

-Valores no Válidos

Se evalúa también para las salidas:

-Valores Válidos

-Valores no Válidos

Entradas aplicarDescuento(): VV: Porcentaje positivo. De 1 a 99.

VN: De 0 a Integer.MIN_VALUE

VN: De 100 a Integer.MAX_VALUE

Tres casos de prueba por cada clase de equivalencia tanto Válida como de no Válida.

Entradas setIVA(): VV:21

VV:10

VV:4

VN: Cualquier otro que no sean estos tres valores.

Cuatro casos de prueba por cada clase de equivalencia tanto Válida como de no Válida.

Entrada setPrecio(): VV: De 0.1 a Double.MAX_VALUE

VN: De 0.0 a Double.MAX_VALUE

Dos casos de prueba por cada clase de equivalencia tanto Válida como de no Válida.

Salida aplicarDescuento(): No se estudian en el método aplicarDescuento() porque no devuelven nada, es tipo Void.

Salida setIVA(): No se estudian en el método setPrecio() porque no devuelven nada, es tipo Void.

Salida setPrecio(): No se estudian en el método setIVA() porque no devuelven nada, es tipo Void.

Tendremos nueve casos de prueba en total.

2º. A partir de las clases de equivalencia obtener los casos de prueba de caja negra.

Casos de Prueba		
CE	Caso de prueba	Resultado Esperado

1 -	`Producto p = new Producto();`
	`p.setPrecio(100.0);`
	`p.aplicarDescuento(20);`
	`p.getPrecio();`
		80.0

2 -	`Producto p = new Producto();`
	`p.setPrecio(100.0);`
	`p.aplicarDescuento(-2);`
	`p.getPrecio();`
		100.0

3 -	`Producto p = new Producto();`
	`p.setPrecio(100.0)`
	`p.aplicarDescuento(120)`
	`p.getPrecio();`
		100.0

4 -	`Producto p = new Producto();`
	`p.setIVA(21);`
	`p.getIVA();`
		21

5 -	`Producto p = new Producto();`
	`p.setIVA(10);`
	`p.getIVA();`
		10

6 -	`Producto p = new Producto();`
	`p.setIVA(4);`
	`p.getIVA();`
		4

7 -	`Producto p = new Producto();`
	`p.setIVA(1000)`
	`p.getIVA();`
		21

8 -	`Producto p = new Producto();`
	`p.setPrecio(90.0);`
	`p.getPrecio;`
		90.0

9 -	`Producto p = new Producto();`
	`p.setPrecio(-50.0);`
	`p.getPrecio;`
		0.0

3º. Codificar las pruebas con JUnit para validar el correcto funcionamiento de la clase.

```java
package es.studium.Tarea4;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;


class Test_Caja_Negra
{

    @Test
    void Camino1()
    {
        Producto p = new Producto();
        p.setPrecio(100.0);
        p.aplicarDescuento(20);
        double resultadoReal = p.getPrecio();
        double resultadoEsperado = 80.0;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino2()
    {
        Producto p = new Producto();
        p.setPrecio(100.0);
        /*p.aplicarDescuento(-2);
          Esta línea no se ejecuta, porque según indica el
          enunciado, si se hace una operación no válida no cambian los
          atributos del objeto.*/
        double resultadoReal = p.getPrecio();
        double resultadoEsperado =100.0;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino3()
    {
        Producto p = new Producto();
        p.setPrecio(100.0);
        /*p.aplicarDescuento(120);
          Esta línea no se ejecuta, porque según indica el
          enunciado, si se hace una operación no válida no cambian los
          atributos del objeto.*/
        double resultadoReal = p.getPrecio();
        double resultadoEsperado = 100.0;
        assertEquals(resultadoEsperado, resultadoReal);
    }

```

```java
@Test
    void Camino4()
    {
        Producto p = new Producto();
        p.setIVA(21);
        int resultadoReal = p.getIVA();
        int resultadoEsperado = 21;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino5()
    {
        Producto p = new Producto();
        p.setIVA(10);
        int resultadoReal = p.getIVA();
        int resultadoEsperado = 10;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino6()
    {
        Producto p = new Producto();
        p.setIVA(4);
        int resultadoReal = p.getIVA();
        int resultadoEsperado = 4;
        assertEquals(resultadoEsperado, resultadoReal);
    }
```

```java
@Test
    void Camino7()
    {
        Producto p = new Producto();
        //p.setIVA(1000);
        /*Esta línea no se ejecuta, porque según indica el
          enunciado, si se hace una operación no válida no cambian los
          atributos del objeto.*/
        int resultadoReal = p.getIVA();
        int resultadoEsperado = 21;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino8()
    {
        Producto p = new Producto();
        p.setPrecio(90.0);
        double resultadoReal = p.getPrecio();
        double resultadoEsperado = 90.0;
        assertEquals(resultadoEsperado, resultadoReal);
    }

    @Test
    void Camino9()
    {
        Producto p = new Producto();
        //p.setPrecio(-50.0);
        /*Esta línea no se ejecuta, porque según indica el
          enunciado, si se hace una operación no válida no cambian los
          atributos del objeto.*/
        double resultadoReal = p.getPrecio();
        double resultadoEsperado = 0.0;
        assertEquals(resultadoEsperado, resultadoReal);
    }


}
```

Conclusión:

Ha sido una práctica muy ilustrativa y enriquecedora a vistas de nuestro futuro profesional. Hemos estudiado y aplicado conocimientos de vital importancia como las pruebas de caja blanca, grafos, casos de prueba y pruebas de caja negra con sus correspondientes casos de prueba y la aplicación de las clases JUnit para validar estas.

Tener que documentar todo este proceso es muy positivo a la hora de enseñarnos los métodos de documentación que también deberemos aplicar una vez que estemos trabajando por nuestra cuenta.

A nivel personal ha sido una experiencia muy enriquecedora la cual de seguro he podido aprender mucho y de la que no me olvidaré.

Los trabajos que estamos realizando con Java y el entorno de desarrollo de Eclipse están siendo muy gratificantes, estos conocimientos son muy importantes a la hora de ser profesionales y poder distinguirnos de la competencia una vez que salgamos al mercado laboral.

Bibliografía:

https://campustudium.com/my/

https://spa.myservername.com/black-box-testing-an-depth-tutorial-with-examples