

## TFG-23.Prueba-isPrime(2024.03.13)

### Prueba Test Unitario de la función isPrime

- Esta función se desarrolló para la asignatura de Ingeniería del Software II del grado de Ingeniería Informática de la EHU/UPV.

### Función Inicial:

```
JAVA

/**
 * @param args Set of strings consisting of integers.
 * @return Returns whether the numbers in args are all prime or not.
 * @throws MissingArgumentException When there are no numbers in args.
 * @throws Only1ArgumentException When there is more than 1 number in the args variable.
 * @throws NoPositiveNumberException When there are negative numbers in the args variable.
 */
public static boolean isPrime(String[] args)
    throws MissingArgumentException, Only1ArgumentException, NoPositiveNumberException
{
    if (args == null)
        throw new MissingArgumentException();
    else if (args.length > 1)
        throw new Only1ArgumentException();
    else {
        try {
            float numF = Float.parseFloat(args[0]);
            int num = (int) numF;
            if (num <= 0)
                throw new NoPositiveNumberException();
            else {
                for (int i = 2; i < num; i++)
                    if (num % i == 0)
                        return false;
                return true;
            }
        } catch (NumberFormatException e) {
            throw new NoPositiveNumberException();
        }
    }
}
```

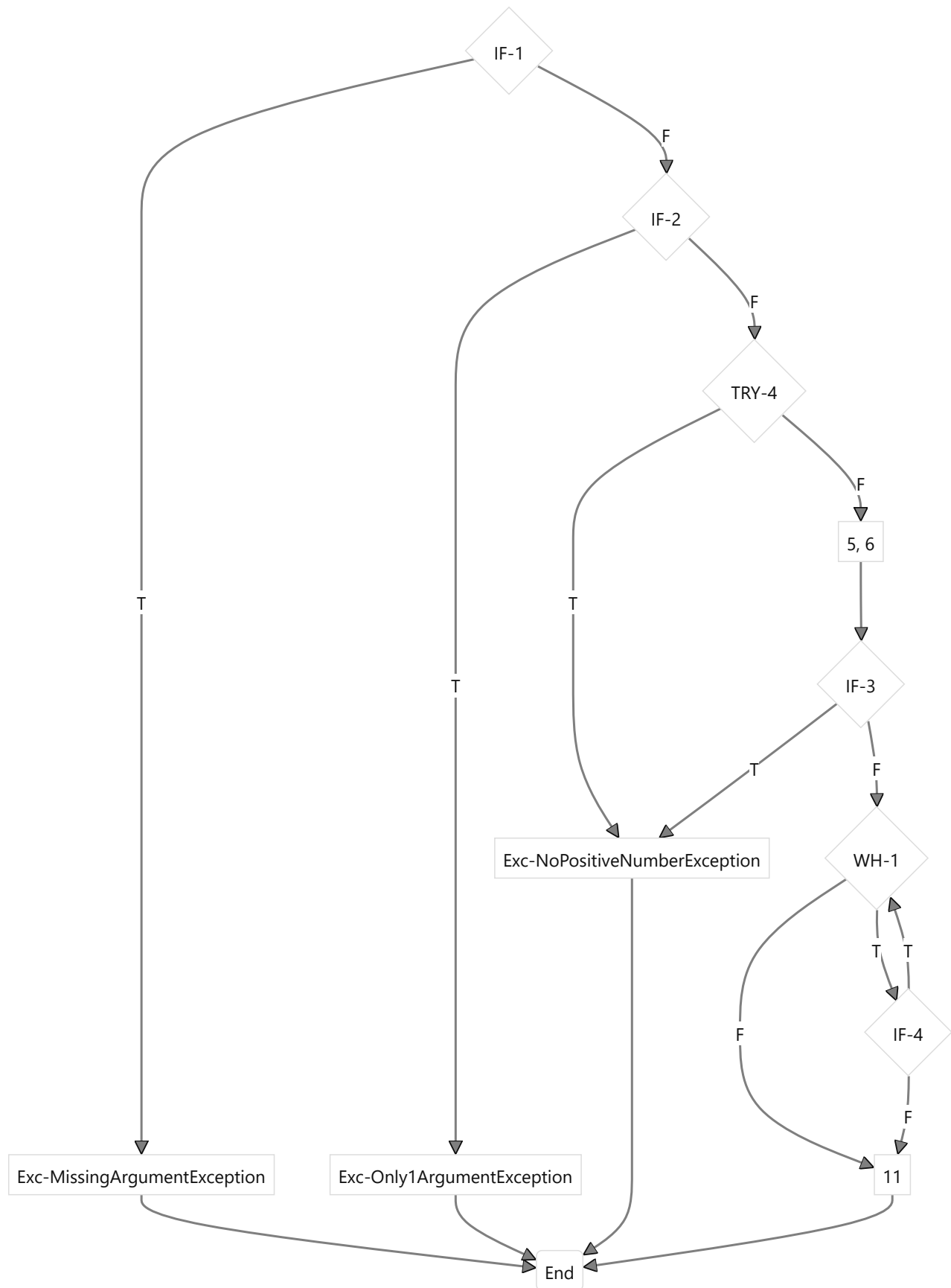
- El resto de clases y funciones auxiliares a esta se tomarán como correctas y validadas. Estas son las siguientes:
  - Clase de excepción: **MissingArgumentException**
    - Se lanza cuando no se introduce ningún argumento en los parámetros.
  - Clase de excepción: **Only1ArgumentException**
    - Se lanza cuando se encuentra más de 1 argumento en el parámetro args.
  - Clase de excepción: **NoPositiveNumberException**
    - Se lanza cuando se encuentran números negativos en el parámetro args.

### Test de Caja Blanca:

- Siguiendo con la metodología de la asignatura II2, he elaborado un set de tests manualmente para comprobar la correcta implementación y funcionamiento de **isPrime**.

## Condition

## Cyclomatic Complexity:



## Test Case Table:

#case	Flow (Conditions)	Condition	Input	Output
1	IF-1(T)	args == null	args = null	Exception: MissingArgumentException

#case	Flow (Conditions)	Condition	Input	Output
2	IF-1(F), IF-2(T)	args.length > 1	args = {4, 12}	Exception: Only1ArgumentException
3	IF-1(F), IF-2(F), IF-3(T)	num < 0	num = -4	Exception: NoPositiveNumberException
4	IF-1(F), IF-2(F), IF-3(F), WH-1{0}	num = 2	num = 2	True
5	IF-1(F), IF-2(F), IF-3(F), WH-1{1} [IF-4(F)]	num = 3	num = 3	True
6	IF-1(F), IF-2(F), IF-3(F), WH-1{1} [IF-4(T)]	num = 4	num = 4	False
7	IF-1(F), IF-2(F), IF-3(F), WH-1+	num > 4	num = 17	True
8	Exception: NoPositiveNumberException	args[0] = NaN.	args[0] = "NaN."	Exception: NoPositiveNumberException

```

✔ isPrimeDWCTest1() calculateMonthDays Function Dynamic White Box Condition Unit Test: 1st Case: 2.0ms
✔ isPrimeDWCTest2() calculateMonthDays Function Dynamic White Box Condition Unit Test: 2nd Case: 1.0ms
✔ isPrimeDWCTest3() calculateMonthDays Function Dynamic White Box Condition Unit Test: 3rd Case: 1.0ms
✔ isPrimeDWCTest4() calculateMonthDays Function Dynamic White Box Condition Unit Test: 4th Case: 1.0ms
✔ isPrimeDWCTest5() calculateMonthDays Function Dynamic White Box Condition Unit Test: 5th Case: 0.0ms
✔ isPrimeDWCTest6() calculateMonthDays Function Dynamic White Box Condition Unit Test: 6th Case: 1.0ms
✔ isPrimeDWCTest7() calculateMonthDays Function Dynamic White Box Condition Unit Test: 7th Case: 0.0ms
✔ isPrimeDWCTest8() calculateMonthDays Function Dynamic White Box Condition Unit Test: 8th Case: 1.0ms

```

- He aquí la cobertura del testing manual creado gracias al programa *Jacoco*:

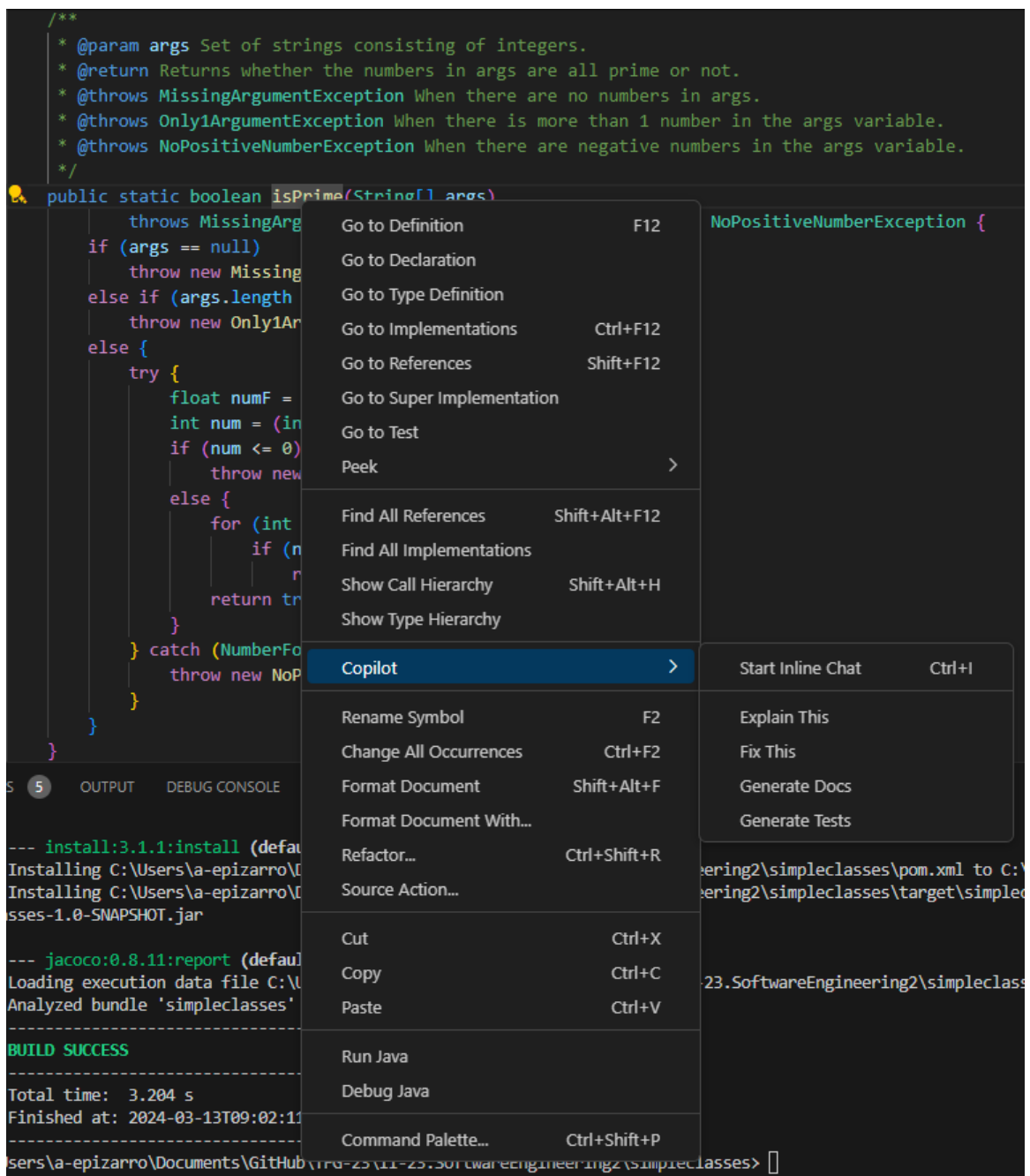
simpleclasses > functions > Functions

### Functions

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getMonthDays(int)		50 %		33 %	2	3	2	4	0	1
Functions()		0 %		n/a	1	1	1	1	1	1
isPrime(String[])		100 %		100 %	0	6	0	14	0	1
invert(String)		100 %		75 %	2	5	0	11	0	1
isLeapYear(int)		100 %		100 %	0	2	0	3	0	1
calculateMonthDays(int, int)		100 %		100 %	0	3	0	3	0	1
Total	7 of 129	94 %	4 of 27	85 %	5	20	3	36	1	6

## Generación Automática con Copilot:

- Voy a generar los test de caja blanca automáticamente mediante GitHub Copilot:



- He aquí los tests y su ejecución:

```

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

import exceptions.InvertException;
import exceptions.NoPositiveNumberException;
import functions.Functions;

public class IsPrimeCopilotTests {

    @Test
    public void testIsLeapYear() {
        Assertions.assertTrue(Functions.isLeapYear(2020));
        Assertions.assertFalse(Functions.isLeapYear(2021));
        Assertions.assertFalse(Functions.isLeapYear(1900));
        Assertions.assertTrue(Functions.isLeapYear(2000));
    }

    @Test
    public void testGetMonthDays() {
        Assertions.assertEquals(31, Functions.getMonthDays(0)); // January
        Assertions.assertEquals(28, Functions.getMonthDays(1)); // February
        Assertions.assertEquals(30, Functions.getMonthDays(3)); // April
        Assertions.assertEquals(30, Functions.getMonthDays(5)); // June
        Assertions.assertEquals(30, Functions.getMonthDays(8)); // September
        Assertions.assertEquals(30, Functions.getMonthDays(10)); // November
    }

    @Test
    public void testCalculateMonthDays() {
        Assertions.assertEquals(31, Functions.calculateMonthDays(2022, 0)); // January
        Assertions.assertEquals(29, Functions.calculateMonthDays(2020, 1)); // February (leap
year)
        Assertions.assertEquals(28, Functions.calculateMonthDays(2021, 1)); // February (non-
leap year)
        Assertions.assertEquals(30, Functions.calculateMonthDays(2022, 3)); // April
        Assertions.assertEquals(30, Functions.calculateMonthDays(2022, 5)); // June
        Assertions.assertEquals(30, Functions.calculateMonthDays(2022, 8)); // September
        Assertions.assertEquals(30, Functions.calculateMonthDays(2022, 10)); // November
    }

    @Test
    public void testIsPrime() throws Exception {
        String[] args1 = { "2" };
        String[] args2 = { "5" };
        String[] args3 = { "10" };
        String[] args4 = { "-7" };
        String[] args5 = { "abc" };

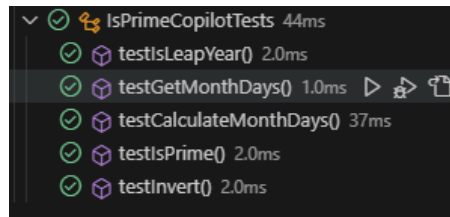
        Assertions.assertTrue(Functions.isPrime(args1));
        Assertions.assertTrue(Functions.isPrime(args2));
        Assertions.assertFalse(Functions.isPrime(args3));
        Assertions.assertThrows(NoPositiveNumberException.class, () ->
Functions.isPrime(args4));
        Assertions.assertThrows(NoPositiveNumberException.class, () ->
Functions.isPrime(args5));
    }
}

```

```

@Test
public void testInvert() throws Exception {
    Assertions.assertEquals(321, Functions.invert("123"));
    Assertions.assertEquals(987654321, Functions.invert("123456789"));
    Assertions.assertThrows(InvertException.class, () -> Functions.invert("0"));
    Assertions.assertThrows(InvertException.class, () -> Functions.invert("-123"));
    Assertions.assertThrows(InvertException.class, () -> Functions.invert("12345678901"));
}
}

```



![[Pasted image 20240313092007.png]]

## Functions

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isPrime(String[])	<div><div></div></div>	83 %	<div><div></div></div>	80 %	2	6	2	14	0	1

```

/**
 * @param args Set of strings consisting of integers.
 * @return Returns whether the numbers in args are all prime or not.
 * @throws MissingArgumentException When there are no numbers in args.
 * @throws Only1ArgumentException When there is more than 1 number in the args variable.
 * @throws NoPositiveNumberException When there are negative numbers in the args variable.
 */
public static boolean isPrime(String[] args)
    throws MissingArgumentException, Only1ArgumentException, NoPositiveNumberException {
    if (args == null)
        throw new MissingArgumentException();
    else if (args.length > 1)
        throw new Only1ArgumentException();
    else {
        try {
            float numF = Float.parseFloat(args[0]);
            int num = (int) numF;
            if (num <= 0)
                throw new NoPositiveNumberException();
            else {
                for (int i = 2; i < num; i++)
                    if (num % i == 0)
                        return false;
                return true;
            }
        } catch (NumberFormatException e) {
            throw new NoPositiveNumberException();
        }
    }
}

```

## Conclusión:

- En el primer intento de generar casos de test de caja blanca se ha olvidado de los casos de argumentos nulos o de más de un argumentos.
- En primera instancia, podemos inducir que Copilot no ha analizado el código, sino que ha tomado casos clásicos de test de número primos y se ha olvidado de las 2 primeras excepciones del código, siendo estas, además, las primeras 2 condiciones.

## Test de Caja Negra:

### Partición de Equivalencia y Valores Límite:

#case	Condition	Input	Output
1	Correct class equivalence partition.	args = {"7"}	True
2	Incorrect class equivalence partition.	args = {"4"}	False

#case	Condition	Input	Output
3	Limit Value Positive: 1	args = {"1"}	True
4	Limit Value Negative: 0	args = {"0"}	Exception: NoPositiveNumberException
5	Exception equivalence partition: Missing Arguments.	args = {"", args = null	Exception: MissingArgumentException
6	Exception equivalence partition: Single Argument Number	args = {"2", "7"}	Exception: Only1ArgumentException
7	Exception equivalence partition: The input is not a number.	args = {"a"}	Exception: NoPositiveNumberException

- Ejecución:

```

✔ isPrimeDBPVTes1() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 1st Case: 1.0ms
✔ isPrimeDBPVTes2() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 2nd Case: 1.0ms
✔ isPrimeDBPVTes3() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 3rd Case: 17ms
✔ isPrimeDBPVTes4() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 4th Case: 1.0ms
✔ isPrimeDBPVTes5() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 5th Case: 1.0ms
✔ isPrimeDBPVTes6() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 6th Case: 2.0ms
✔ isPrimeDBPVTes7() isPrime Function Dynamic Black Box Equivalence Partition and Limit Value Unit Test: 6th Case: 1.0ms

```

## Testing Automático mediante partición de equivalencia y valores límite:

```

/testis Using equivalence partition and boundary analysis methods. Do not use White Box method.
Accept Discard ↺
public static boolean isPrime(String[] args)
    throws MissingArgumentException, Only1ArgumentException, NoPositiveNumberException {
    if (args == null)
        throw new MissingArgumentException();
    else if (args.length > 1)
        throw new Only1ArgumentException();
    else {
        try {

```

- Antes de mostrar todos los test hay que retocar algunos ya que lanzas excepciones y en las pruebas o bien haces aserciones de excepciones o haces try and catch:

```
// Antes:
@Test
void testIsPrimeWithPrimeNumber() throws Exception {
    String[] args = { "7" };
    assertTrue(Functions.isPrime(args));
}

@Test
void testIsPrimeWithNonPrimeNumber() throws Exception {
    String[] args = { "10" };
    assertFalse(Functions.isPrime(args));
}

// Después:
@Test
void testIsPrimeWithPrimeNumber() {
    String[] args = { "7" };
    try {
        assertTrue(Functions.isPrime(args));
    } catch (Exception e) {
        fail("Exception should not be thrown");
    }
}

@Test
void testIsPrimeWithNonPrimeNumber() {
    String[] args = { "10" };
    try {
        assertFalse(Functions.isPrime(args));
    } catch (Exception e) {
        fail("Exception should not be thrown");
    }
}
```

- He aquí todas las pruebas que ha generado Copilot automáticamente:



```
import org.junit.jupiter.api.Test;

import exceptions.MissingArgumentException;
import exceptions.NoPositiveNumberException;
import exceptions.Only1ArgumentException;
import functions.Functions;

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

class IsPrimeCopilotBBTests {

    @Test
    void testIsPrimeWithPrimeNumber() {
        String[] args = { "7" };
        try {
            assertTrue(Functions.isPrime(args));
        } catch (Exception e) {
            fail("Exception should not be thrown");
        }
    }

    @Test
    void testIsPrimeWithNonPrimeNumber() {
        String[] args = { "10" };
        try {
            assertFalse(Functions.isPrime(args));
        } catch (Exception e) {
            fail("Exception should not be thrown");
        }
    }

    @Test
    void testIsPrimeWithNegativeNumber() {
        String[] args = { "-5" };
        assertThrows(NoPositiveNumberException.class, () -> Functions.isPrime(args));
    }

    @Test
    void testIsPrimeWithZero() {
        String[] args = { "0" };
        assertThrows(NoPositiveNumberException.class, () -> Functions.isPrime(args));
    }

    @Test
    void testIsPrimeWithInvalidInput() {
        String[] args = { "abc" };
        assertThrows(NoPositiveNumberException.class, () -> Functions.isPrime(args));
    }

    @Test
    void testIsPrimeWithMissingArgument() {
        String[] args = null;
        assertThrows(MissingArgumentException.class, () -> Functions.isPrime(args));
    }

    @Test
    void testIsPrimeWithExtraArgument() {
```

```
String[] args = { "5", "10" };  
assertThrows(Only1ArgumentException.class, () -> Functions.isPrime(args));  
}  
}
```

## Conclusión:

- Ha introducido un test redundante al utilizar el método de valores límites. En esta función el valor límite negativo es "0", y lo contempla, pero también prueba con el valor "-5" que es redundante.
- Para puntualizar, queda también, que si hubiera utilizado el método de valores límites, el positivo sería el "1", el cual es el umbral menor positivo. En realidad este valor también sería redundante porque ya prueba un valor positivo. Pero si somos estrictos con la metodología de valores límites, este valor umbral también habría que probarlo.