Pruebas de caja Blanca con JUnit

1. Vamos a aplicar pruebas de caja blanca a una calculadora con JUnit.

   Te dejo un código sencillo en java de una calculadora:

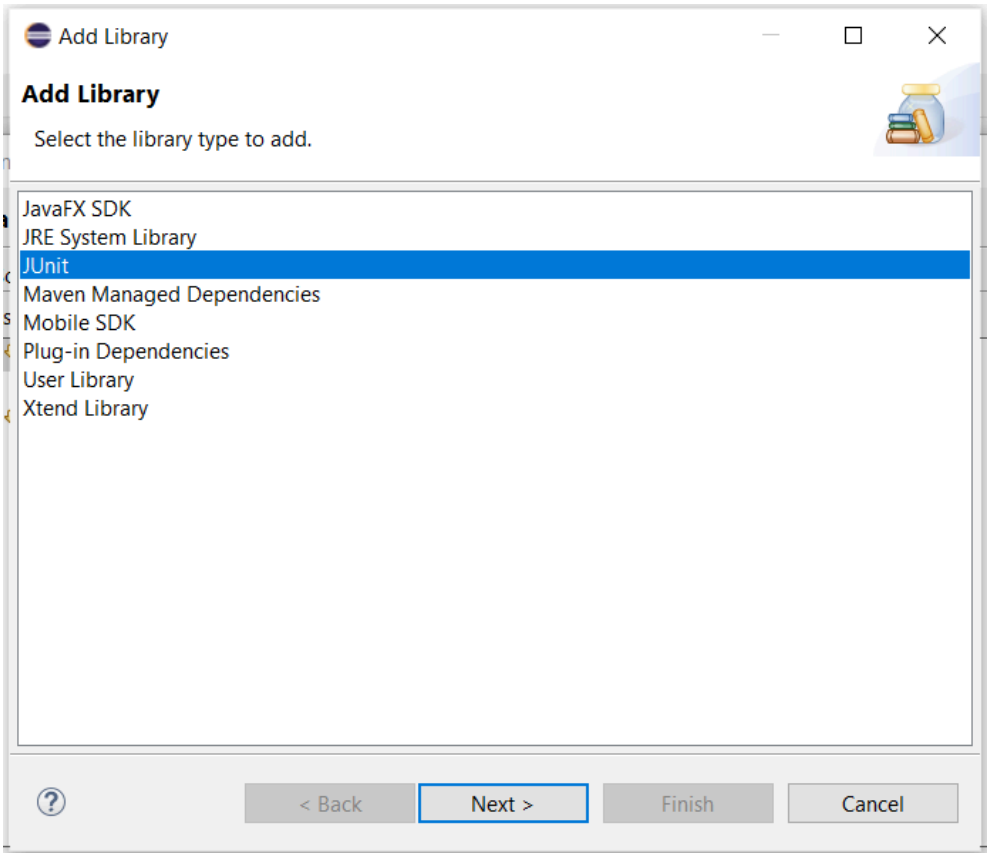
```
public class Calculadora {

    // Suma dos números
    public int sumar(int a, int b) {
        return a + b;
    }

    // Resta dos números
    public int restar(int a, int b) {
        return a - b;
    }

    // Multiplica dos números
    public int multiplicar(int a, int b) {
        return a * b;
    }

    // Divide dos números con control de división por cero
    public double dividir(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("No se puede dividir por cero");
        }
        return (double) a / b;
    }

    // Calcula el factorial de un número con control de entrada negativa
    public int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("No se puede calcular el factorial de un
número negativo");
        }
        int resultado = 1;
        for (int i = 1; i <= n; i++) {
            resultado *= i;
```

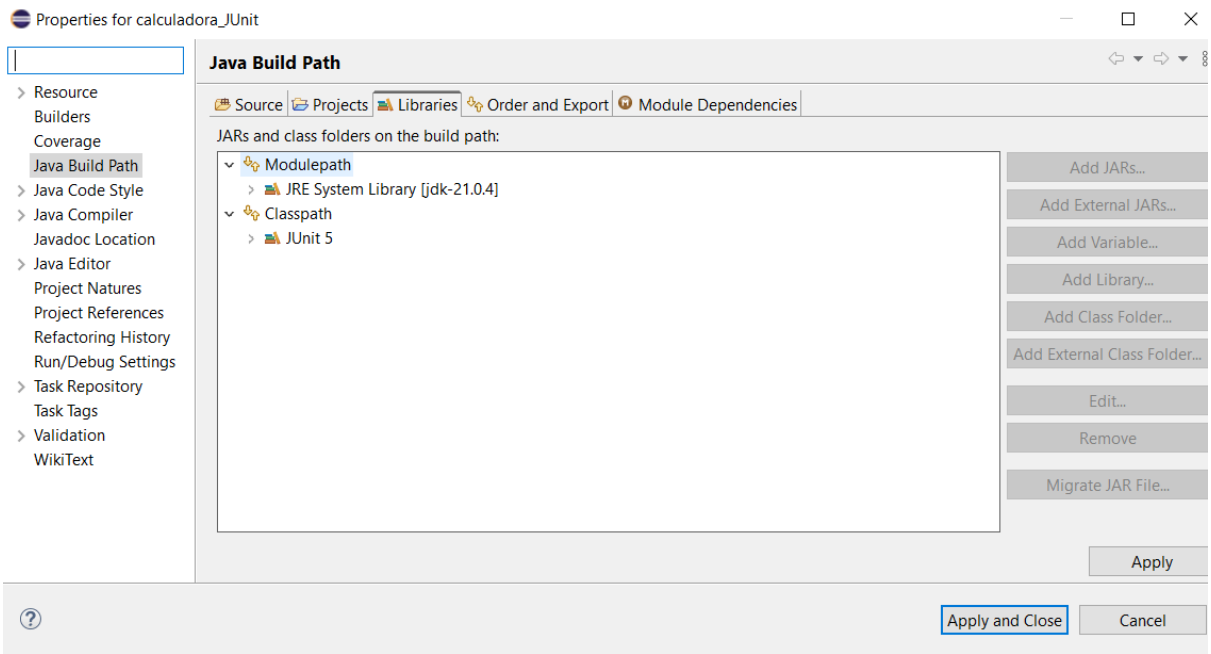
```
        }
        return resultado;
    }

    // Determina si un número es primo
    public boolean esPrimo(int n) {
        if (n < 2) return false;
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    }

}
```

2. **Identifica los caminos lógicos en cada método realizando el grafo**.

   Puedes usar esta herramienta.

3. Añade la librería de JUnit a tu proyecto:





4. He creado **esta clase de prueba** CalculadoraTest con JUnit para probar la calculadora.

   Ejecuta con JUnit y no debería detectar ningún error.

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

public class CalculadoraTest {

    private final Calculadora calc = new Calculadora();

    @Test
    public void testSumar() {
        assertEquals(5, calc.sumar(2, 3));
        assertEquals(-1, calc.sumar(-2, 1));
    }

    @Test
    public void testRestar() {
        assertEquals(1, calc.restar(3, 2));
        assertEquals(-3, calc.restar(-2, 1));
    }

    @Test
    public void testMultiplicar() {
```

```
        assertEquals(6, calc.multiplicar(2, 3));
        assertEquals(0, calc.multiplicar(0, 5));
    }

    @Test
    public void testDividir() {
        assertEquals(2.0, calc.dividir(6, 3));
        assertThrows(IllegalArgumentException.class, () -> calc.dividir(5, 0));
    }

    @Test
    public void testFactorial() {
        assertEquals(120, calc.factorial(5));
        assertEquals(1, calc.factorial(0));
        assertThrows(IllegalArgumentException.class, () -> calc.factorial(-1));
    }

    @Test
    public void testEsPrimo() {
        assertTrue(calc.esPrimo(7));
        assertFalse(calc.esPrimo(4));
        assertFalse(calc.esPrimo(1));
    }
}
```

5. Imagina que haces un **código de pruebas mal hecho**, prueba este sobre el código de la calculadora, **¿qué ocurre?**

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class CalculadoraTest_conerrores {
    private final Calculadora calc = new Calculadora();
    @Test
    public void testSumar() {
        assertEquals(6, calc.sumar(2, 3)); // ERROR: El resultado correcto es 5
    }
    @Test
    public void testRestar() {
        assertEquals(2, calc.restar(3, 2)); // ERROR: El resultado correcto es 1
    }
    @Test
    public void testMultiplicar() {
        assertEquals(12, calc.multiplicar(2, 3)); // ERROR: El resultado correcto es 6
    }
    @Test
    public void testDividir() {
        assertEquals(3.0, calc.dividir(6, 3)); // ERROR: El resultado correcto es 2.0
```

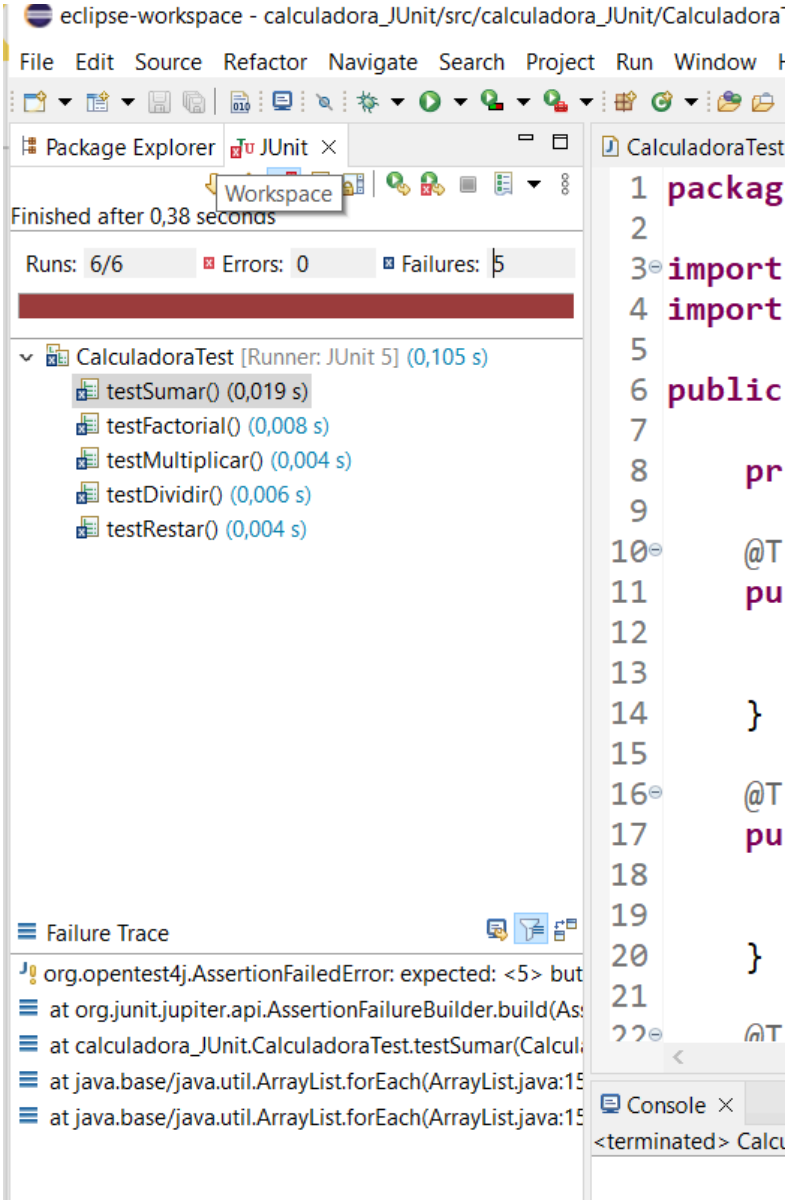
```
    }
    @Test
    public void testFactorial() {
        assertEquals(24, calc.factorial(5)); // ERROR: El resultado correcto es 120
    }
    @Test
    public void testEsPrimo() {
        assertTrue(calc.esPrimo(9)); // ERROR: 9 no es primo
    }
    @Test
    public void testDividirPorCero() {
        assertDoesNotThrow(() -> calc.dividir(5, 0)); // ERROR: Debe lanzar una
excepción
    }
}
```

6. Ahora vamos a hacer un **código mal hecho de la calculadora**, te dejo uno con errores muy básicos:

```
public class Calculadora_errores {
        // Suma dos números (ERROR: retorna resta en lugar de suma)
  public int sumar(int a, int b) {
      return a - b;
  }
  // Resta dos números (ERROR: retorna suma en lugar de resta)
  public int restar(int a, int b) {
      return a + b;
  }
  // Multiplica dos números (ERROR: siempre retorna 0)
  public int multiplicar(int a, int b) {
      return 0;
  }
  // Divide dos números (ERROR: no maneja división por cero correctamente)
  public double dividir(int a, int b) {
      return (double) a / b; // No lanza excepción si b == 0
  }
  // Calcula el factorial (ERROR: inicia con 0 en lugar de 1)
  public int factorial(int n) {
      if (n < 0) {
          throw new IllegalArgumentException("No se puede calcular el factorial de un
número negativo");
      }
      int resultado = 0; // ERROR aquí, debe ser 1
      for (int i = 1; i <= n; i++) {
          resultado *= i;
      }
```

```
        return resultado;
    }
    // Determina si un número es primo (ERROR: considera todos los números impares como primos)
    public boolean esPrimo(int n) {
        if (n < 2) return false;
        if (n % 2 != 0) return true; // ERROR aquí, ignora otros factores
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    }
}
```

7. Pásale el test bien hecho de JUnit a este nuevo código. Deberías obtener los siguientes fallos:



Analiza los errores, **¿serías capaz de hacer un buen test de uno de tus códigos?**

INVESTIGA: patrón **Arrange – Act – Assert (AAA)** es una forma de estructurar un test unitario para que sea claro y mantenible.

1. Arrange (Preparar)
Se prepara el contexto del test.

- Crear objetos.
- Definir datos de entrada.

- Configurar mocks si los hay.

2. Act (Actuar)
   - Se ejecuta la acción que se quiere probar.
   - Se llama al método.

3. Assert (Comprobar)
   - Se verifica el resultado.
   - Comparar el resultado esperado con el real.
   - Comprobar excepciones o cambios de estado.

Ejemplo sencillo:

```
// Arrange
Calculadora calc = new Calculadora();

// Act
int resultado = calc.dividir(10, 2);

// Assert
assertEquals(5, resultado);
```