

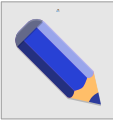
Proba de Desenvolvemento de Interfaces: UD8

8.1 - CA3.1:

20% (ME)

8.2 - CA3.2:

20% (ME)



Apellidos, Nombre: Agra Casal, Rubén

Fecha: 17/03/2025

Unidad 8: Realización de pruebas

CA2.1 - Empaquetáronse os compoñentes que require a aplicación. (20% - ME)

8.1 Dado el siguiente código en Java, indica qué estrategia de pruebas seguirías para minimizar la posibilidad de existencia de errores:

```
package calculator;
```

```
public class Calculator {  
    public double divide(double operand1, double operand2) {  
        return operand1 / operand2;  
    }  
}
```

Con este código realizaría pruebas funcionales utilizando valores límite, con valores que estén dentro de los límites al azar y por último de rendimiento.

Con los valores límites se podría hacer utilizando las constantes del tipo Double que emplea el método de la siguiente forma:

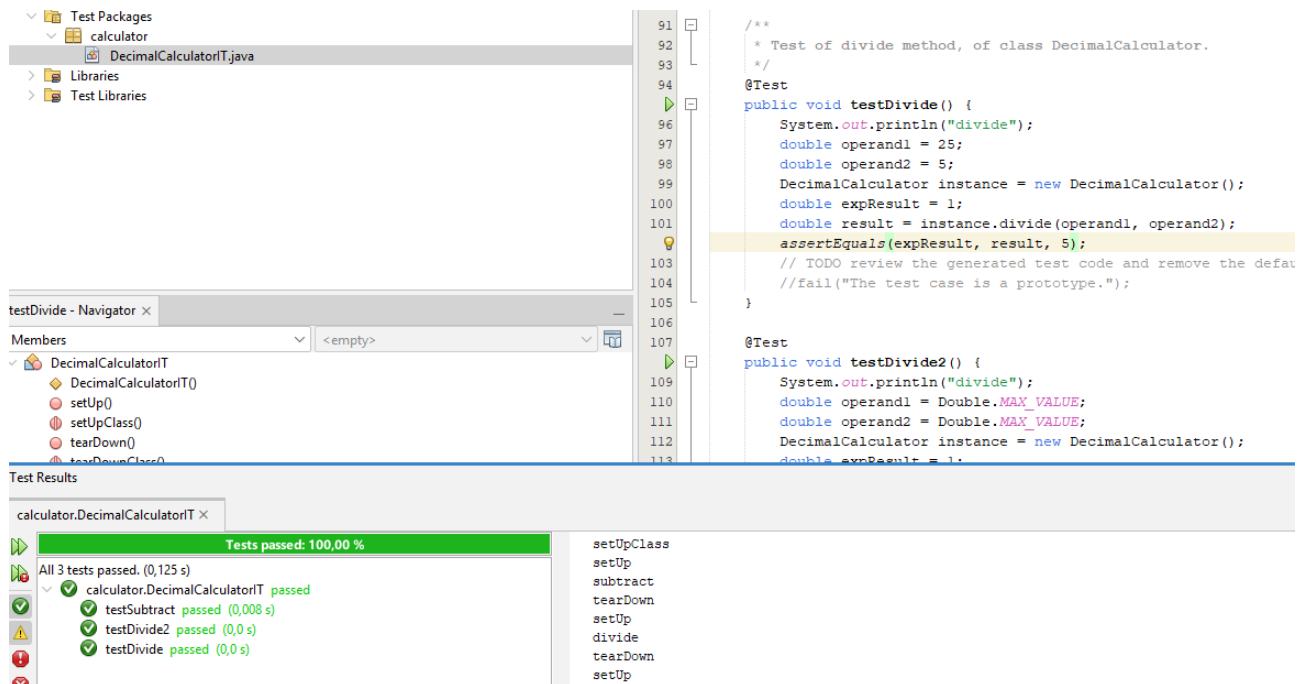
```

    */
@Test
public void testDivide() {
    System.out.println("divide");
    double operand1 = Double.MIN_VALUE;
    double operand2 = Double.MIN_VALUE;
    DecimalCalculator instance = new DecimalCalculator();
    double expectedResult = 1;
    double result = instance.divide(operand1, operand2);
    assertEquals(expResult, result, 0);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}

@Test
public void testDivide2() {
    System.out.println("divide");
    double operand1 = Double.MAX_VALUE;
    double operand2 = Double.MAX_VALUE;
    DecimalCalculator instance = new DecimalCalculator();
    double expectedResult = 1;
    double result = instance.divide(operand1, operand2);
    assertEquals(expResult, result, 0);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}
}

```

Luego para valores entre los los valores límite, se pondría por ejemplo 25, 5 y como resultado daría 5.



The screenshot displays an IDE with the following components:

- Project Explorer:** Shows a package structure with 'Test Packages', 'calculator', and 'DecimalCalculatorIT.java'.
- Code Editor:** Contains Java code for a test class. The visible code includes:


```

91  /**
92   * Test of divide method, of class DecimalCalculator.
93   */
94  @Test
95  public void testDivide() {
96      System.out.println("divide");
97      double operand1 = 25;
98      double operand2 = 5;
99      DecimalCalculator instance = new DecimalCalculator();
100     double expectedResult = 1;
101     double result = instance.divide(operand1, operand2);
102     assertEquals(expResult, result, 5);
103     // TODO review the generated test code and remove the default
104     //fail("The test case is a prototype.");
105 }
106
107 @Test
108 public void testDivide2() {
109     System.out.println("divide");
110     double operand1 = Double.MAX_VALUE;
111     double operand2 = Double.MAX_VALUE;
112     DecimalCalculator instance = new DecimalCalculator();
113     double expectedResult = 1;
      
```
- testDivide - Navigator:** Shows a list of members for 'DecimalCalculatorIT', including 'setUp()', 'setUpClass()', 'tearDown()', and 'tearDownClass()'.
- Test Results:** A panel showing the outcome of the tests. It indicates 'Tests passed: 100,00 %' and lists the following results:
 - All 3 tests passed. (0,125 s)
 - calculator.DecimalCalculatorIT passed
 - testSubtract passed (0,008 s)
 - testDivide2 passed (0,0 s)
 - testDivide passed (0,0 s)
- Method List:** A vertical list of methods on the right side of the Test Results panel: 'setUpClass', 'setUp', 'subtract', 'tearDown', 'setUp', 'divide', 'tearDown', and 'setUp'.

También podríamos poner en el operan2 un 0 para forzar el error ya que no se podría dividir entre ese número.

Por último para el rendimiento deberíamos poner el método por ejemplo en un bucle "for" y que hiciese la operación un número elevado de veces para ver cuanto tiempo tarda en realizarse y ver como el programa reacciona ante esto. Este tipo de prueba sería muy útil si el método se utilizase una gran cantidad de veces en poco tiempo.

CA3.2 - Realizáronse probas de integración dos elementos. (20% - ME)

8.2 Suponiendo que acabas de implementar la clase Service que gestiona los datos de una reparación y que te encuentras implementando la clase Computer que hace uso de la clase Service, indica la estrategia de pruebas que seguirías para minimizar la posibilidad de error en la integración de las dos clases.

Lo primero que haría serían unas **pruebas unitarias** de la clase que acabamos de crear. Para esto crearemos una clase “main” de test para comprobar que los “getters”, “setters”, constructores y el resto de métodos de la clase Service para ver que funcionan correctamente.

Una vez hecho esto, se pasaría a realizar **pruebas de integración** para comprobar que la interrelación entre Service y Computer no da errores. Para realizar este tipo de pruebas se puede utilizar dos estrategias diferentes la ascendente y la descendente.

Para este caso utilizaría la estrategia **descendente** por la forma en la que se ha creado (Primero Computer y luego añadimos Service).

Para ello, se utilizaría la clase Computer como módulo controlador de la prueba y antes de implementar la clase Service, utilizar un módulo de resguardo. Para probar esto bien podríamos hacer por ejemplo en la clase Computer una lista de objetos tipo Service y comprobar que a esta lista podemos añadir, editar y borrar de forma correcta. Si todo funciona bien, se sustituiría el módulo de resguardo por el real.