Proyecto pruebas de software

2º Desarrollo de aplicaciones multiplataforma

Rubén Agra Casal

Fecha: 20/03/2025

Sumario

Índice de figuras

1. Generación y ejecución de pruebas de software unitarias automatizadas

Para poder generar estas pruebas, el entorno de desarrollo NetBeans nos proporciona una herramienta para poder generarlas de forma muy sencilla.

Para esto, pulsaremos botón derecho en la clase a probar → “Tools” → “Create/Update Tests”.

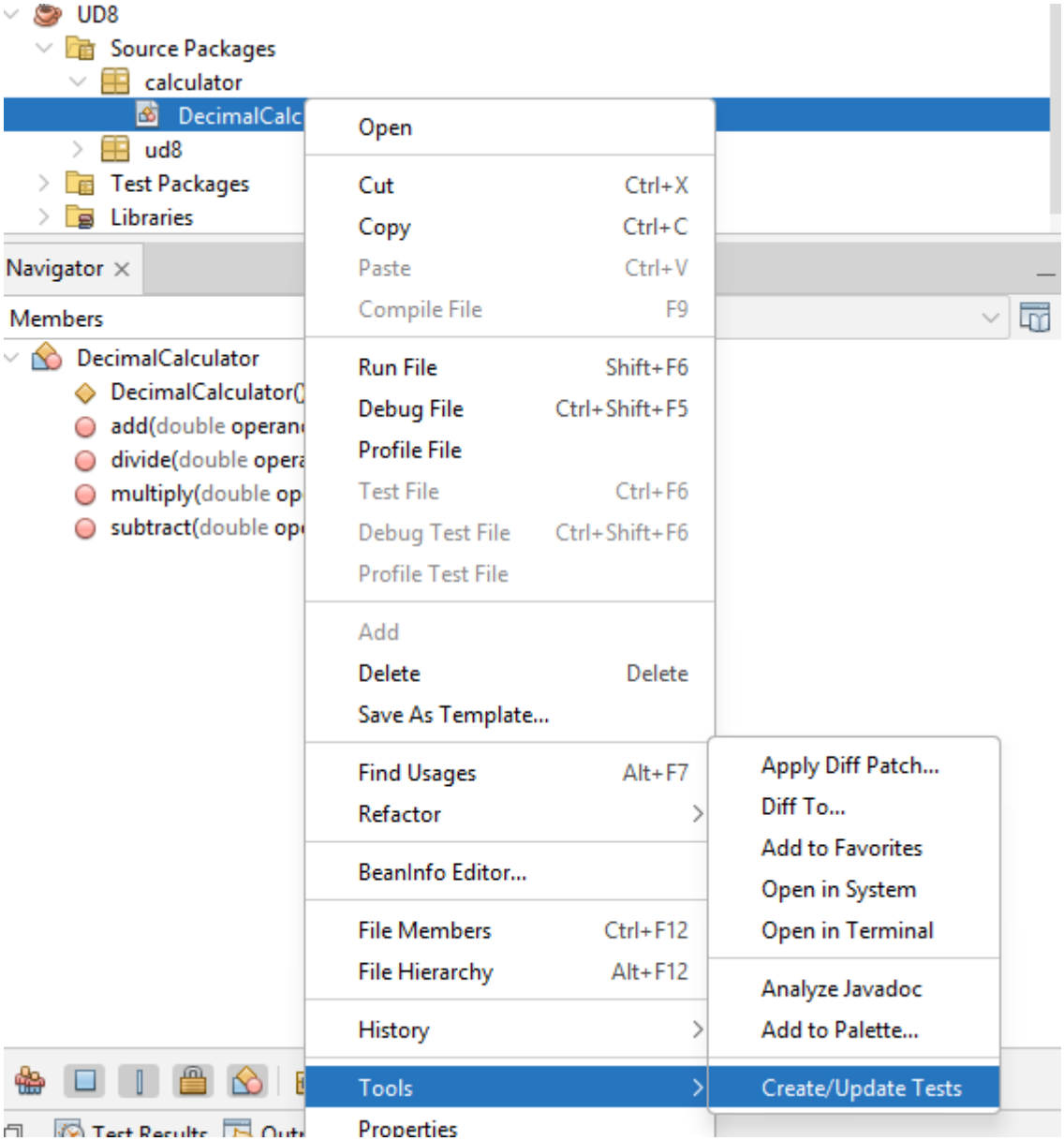

Figura 1: Generación de pruebas automatizadas

Deberemos marcar en la siguiente ventana como framework JUnit4.

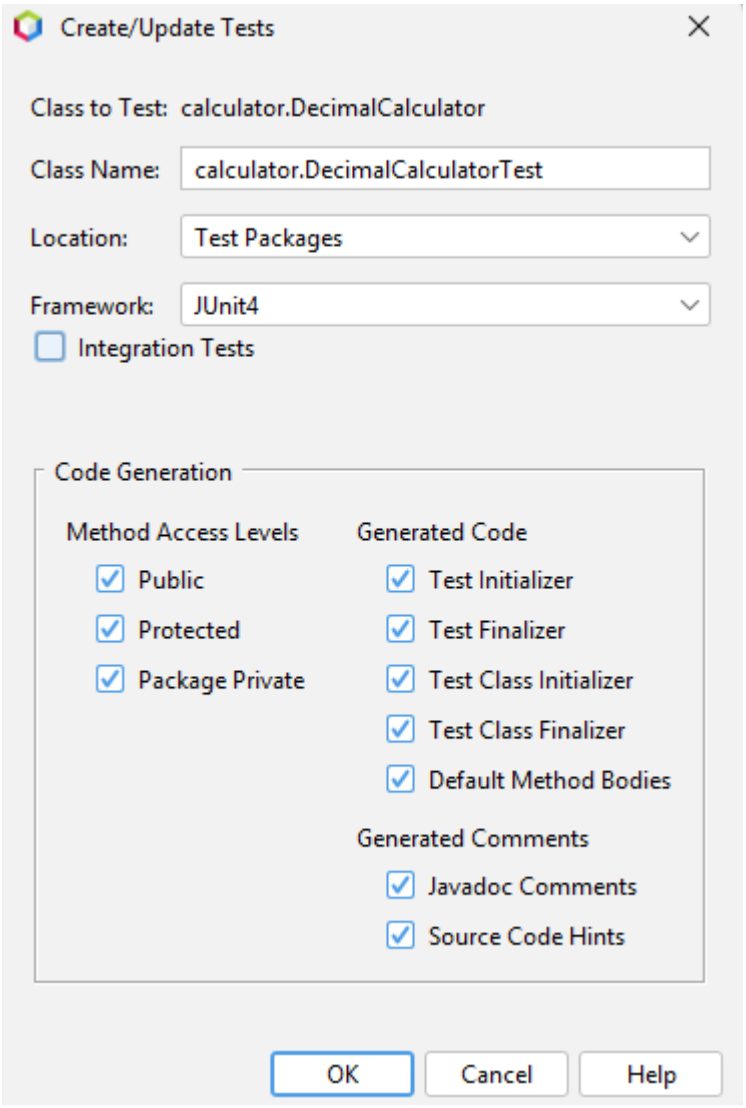

Figura 2: Framework JUnit4

Con esto, nos generará una clase con varios métodos que funcionarán como test (@Test).

Es importante que comentemos o eliminemos la línea final de cada test porque va a generar un fallo aunque el resultado esté bien.

```
 */
@Test
public void testAdd() {
    System.out.println("add");
    double operand1 = 10000.001;
    double operand2 = 0.000000001;
    DecimalCalculator instance = new DecimalCalculator();
    double expResult = 10000.001000001;
    double result = instance.add(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.");
}

/**
```

Figura 3: Línea fail();

Una vez hecho esto, si ejecutamos (Pulsamos con el botón derecho sobe la clase de test y luego pulsaremos en “Run File”, luego pulsaremos en “Test Results” en la parte inferior) el programa nos saldrán que todos las pruebas han salido de forma correcta.

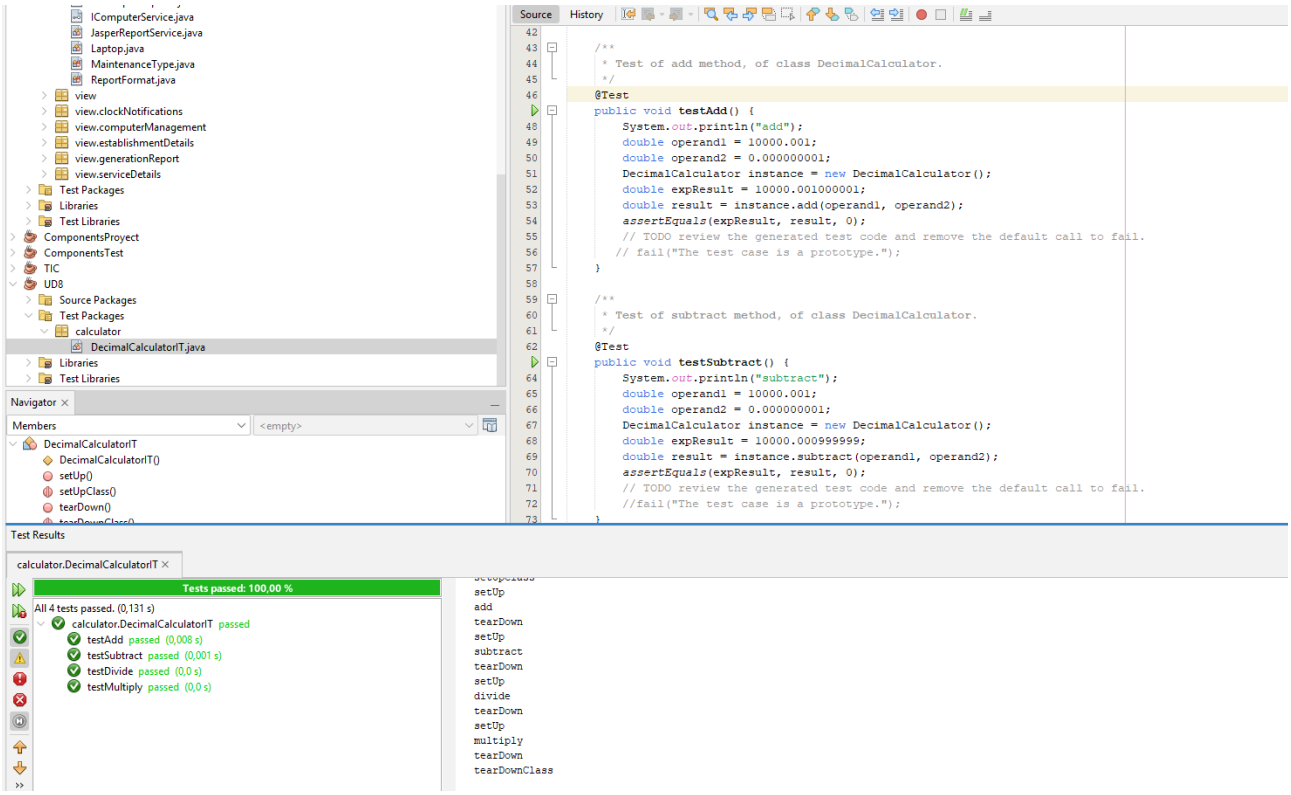

Figura 4: Pruebas superadas

1.1 Pruebas unitarias

Las únicas pruebas unitarias que podemos hacer en este caso sería las de condiciones límite, ya que no estamos trabajando ni con interfaces ni con estructura de datos. Tampoco podríamos comprobar distintos caminos, ya que el método no ofrece esta posibilidad (no se encuentran ni condiciones como “if” ni bucles como “for” o “while”).

Por lo tanto, probaremos utilizando las condiciones límite utilizando las constantes del tipo de objeto que utiliza el método (Double.MAX_VALUE y Double.MIN_VALUE).

```
     */
    @Test
    public void testDivide() {
        System.out.println("divide");
        double operand1 = Double.MIN_VALUE;
        double operand2 = Double.MIN_VALUE;
        DecimalCalculator instance = new DecimalCalculator();
        double expResult = 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 expResult = 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.");
    }

}
```

Figura 5: Valores límite

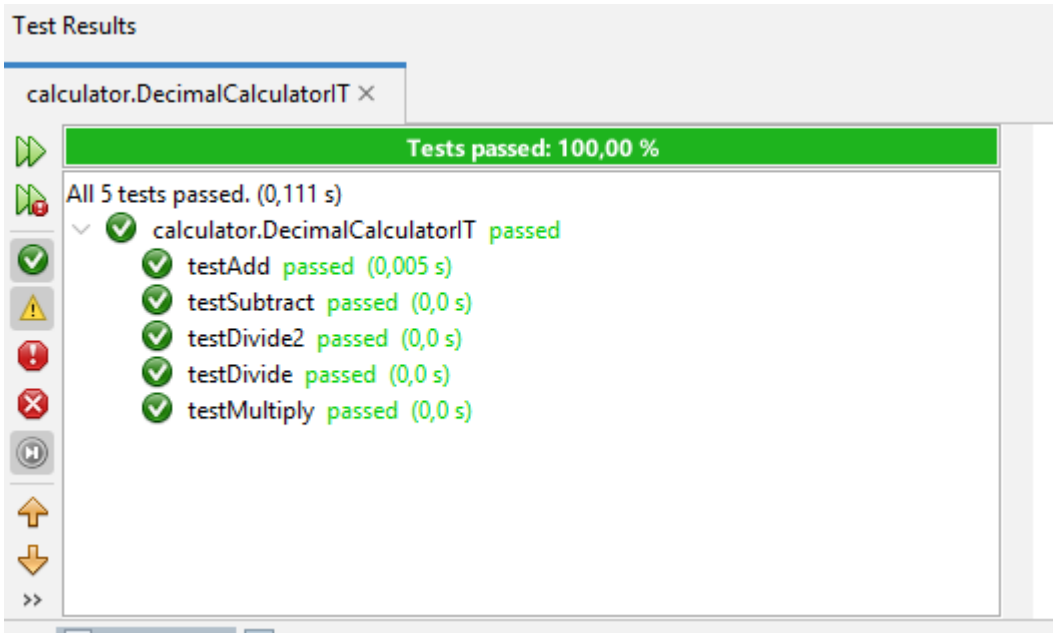

Figura 6: Pruebas con valores límite superadas

1.2 Pruebas de regresión

En el caso de que alguien modifique el código, debemos realizar las conocidas “Pruebas de regresión.

En el supuesto caso de que alguien hubiese modificado el código de esta manera:

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

Las pruebas fallarían:

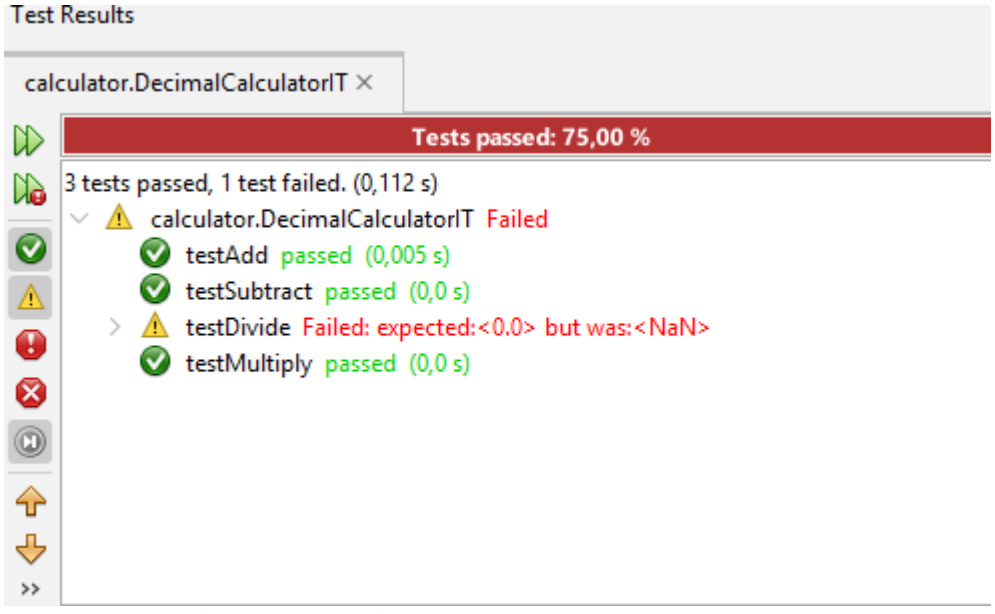

Figura 7: Prueba no superada

Ante estos casos, debemos cambiar el código lo antes posible antes de que muchos usuarios se topen con el error, ya que esto podría provocar un descontento general y una desconfianza hacia nuestro trabajo.

Por este motivo, es muy importante siempre realizar este tipo de pruebas una vez que el código se haya modificado, ya que pueden surgir errores que se nos haya pasado y que pasen a la versión final del programa.

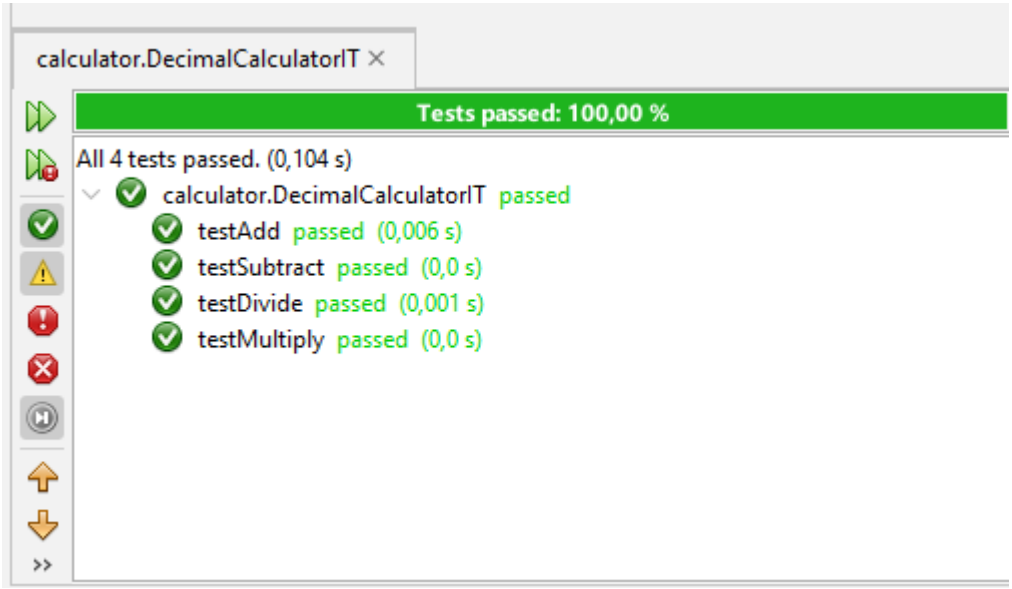

Figura 8: Prueba de regresión realizada y superada

1.3 Pruebas de volumen y estrés

En casos en los que el método se vaya a utilizar una gran cantidad de veces, es importante que realicemos pruebas de este tipo para comprobar si la aplicación es eficiente y podría generar reacciones inesperadas en el programa. Para poder probar esto podríamos hacer que el método se repita una gran cantidad de veces utilizando un bucle por ejemplo y ver cuanto tarda en finalizar.

1.4 Pruebas de seguridad

Para este caso no es necesario realizar este tipo de pruebas porque no estamos tratando con datos confidenciales, privados o sensibles como contraseñas, información de usuarios, etc.

En el caso de que tuviésemos esto, deberíamos hacer pruebas para hacer que nuestro código sea más robusto y asegurar la seguridad del código. Para probar esto podríamos hacer si el código está bien encapsulado, si las contraseñas son correctas, etc.

1.5 Pruebas de uso de recursos

Para medir tiempos debemos tener en cuenta el “overhead”o la sobrecarga, que es el tiempo que tarda el programa en calcular el tiempo. Esto es importante tenerlo en cuenta, porque cuando hagamos pruebas de este tipo, nos darán un valor real por lo que debemos restarle al supuesto valor real el “overhead” para que sea una prueba correcta.

```
 */
public class Main {
    public static void main(String[] args) {
        System.out.println("DC");
        DecimalCalculator dc = new DecimalCalculator();

        long testTimer = System.nanoTime();
        long startTimer = System.nanoTime();

        for(int i = 0; i < 1000000000; i++) {
            dc.divide(1.0, 1.0);
        }

        long endTime = System.nanoTime();
        long overhead = startTimer - testTimer; // Calculamos la sobrecarga
        long executionTime = (endTime - startTimer) - overhead; // Restar sobrecarga

        System.out.println("Done: " + executionTime/1000000+"ms");
    }
}
```

Figura 9: Prueba de uso de recursos

En el código lo vamos haciendo es recoger marcas de tiempo con el método **nanoTime(),** concretamente recogemos 2 antes del bucle y una al finalizar el bucle.

Luego calculamos el overhead restando la segunda marca de tiempo por la primera.

Por último, calculamos el tiempo de ejecución restando la marca de tiempo al finalizar el bucle menos la de inicio menos el overhead calculado anteriormente.