**Apellidos, Nombre: Amado Cibreiro, Andrés**

## 8. Proyecto pruebas de software

*CA3.7 Documentouse a estratexia de probas e os resultados obtidos. (20%)*

8.1 Analiza el código fuente “DecimalCalculator” que se ha facilitado para este proyecto y crea una batería pruebas de software unitarias automatizada y ejecútala. Muestra el código fuente de la clase de pruebas que se ha generado de forma automática, así como el resultado de la ejecución de las pruebas.

8.2 (Pruebas unitarias) Modifica el código fuente de las pruebas que se ha autogenerado para tener en cuenta, si es que procede, lo siguiente:

- Interfaz de módulo correspondiente.
- Impacto de los datos globales sobre el módulo.
- Estructuras de datos en el módulo.
- Las condiciones límite.
- Los distintos caminos de ejecución de las estructuras de control.

*CA3.3 Realizáronse probas de regresión. (10%)*

8.3 (Pruebas de regresión) Un desarrollador ha modificado el código del método divide de la siguiente forma:

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

Implementa este cambio en el proyecto y realiza las correspondientes pruebas de regresión. En el caso de que haya errores de regresión, implementa una solución sabiendo que sólo se puede modificar el cuerpo del módulo e indica qué sentencias han sido modificadas.

*CA3.4 Realizáronse probas de volume e estrés. (10%)*

8.4 (Pruebas de volumen y estrés) Investiga si para el módulo dado sería necesario realizar pruebas de volumen y estrés. Investiga en qué casos sería necesario implementarlas.

*CA3.5 Realizáronse probas de seguridade. (10%)*

8.5 (Pruebas de seguridad) Investiga si para el módulo dado sería necesario realizar pruebas de seguridad. Investiga en qué casos sería necesario implementarlas.

*CA3.6 Realizáronse probas de uso de recursos por parte da aplicación. (10%)*

8.6 (Pruebas de uso de recursos) Investiga cuando tiempo tarda el programa en realizar 1.000.000.000 de divisiones consecutivas

Entrega el proyecto en formato pdf empleando la siguiente nomenclatura: UD8_Apellido1_Apellido2_Nombre.pdf

# 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”.

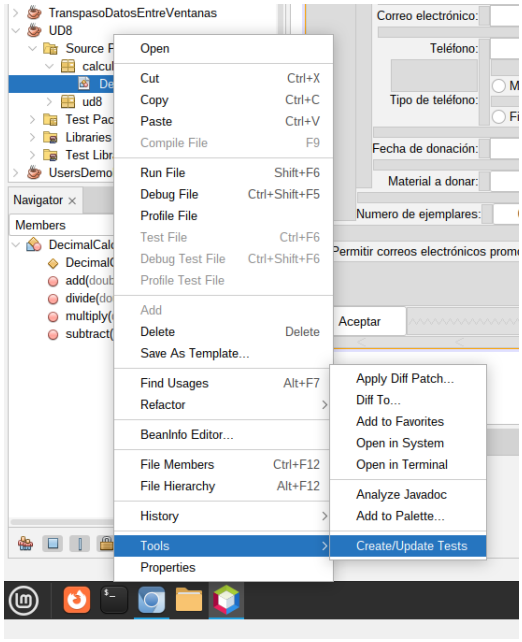


*Figura 1: Generación de pruebas automatizadas*

Deberemos marcar en la siguiente ventana como framework JUnit4.

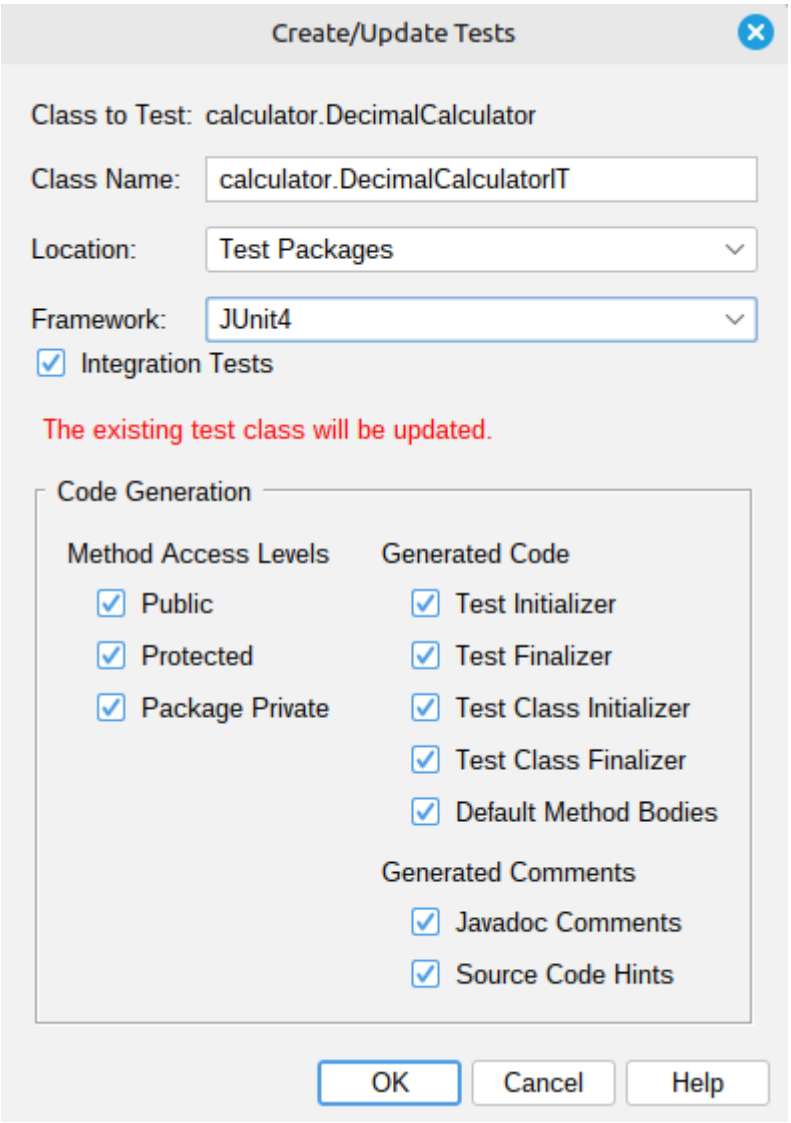


*Figura 2: Framework JUnit4*

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 = 0.0;
    double operand2 = 0.0;
    DecimalCalculator instance = new DecimalCalculator();
    double expResult = 0.0;
    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: Comentar linea 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.

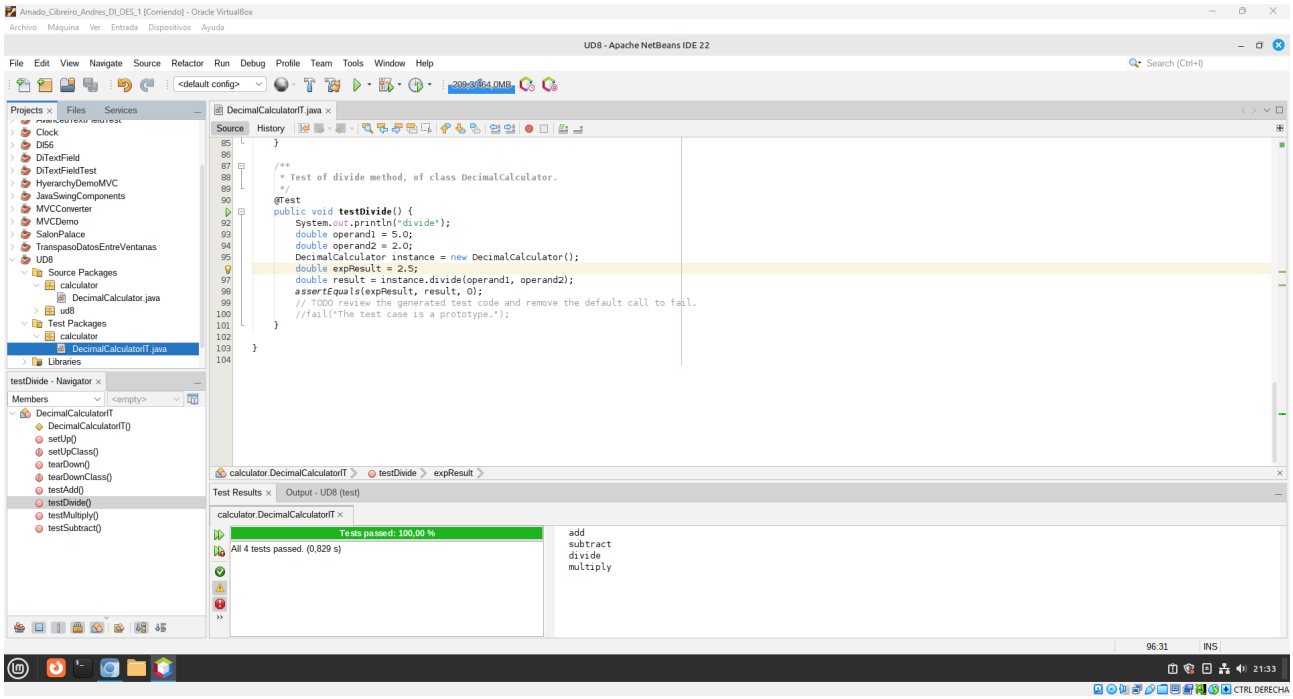


*Figura 4: Test pasado*

## 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).

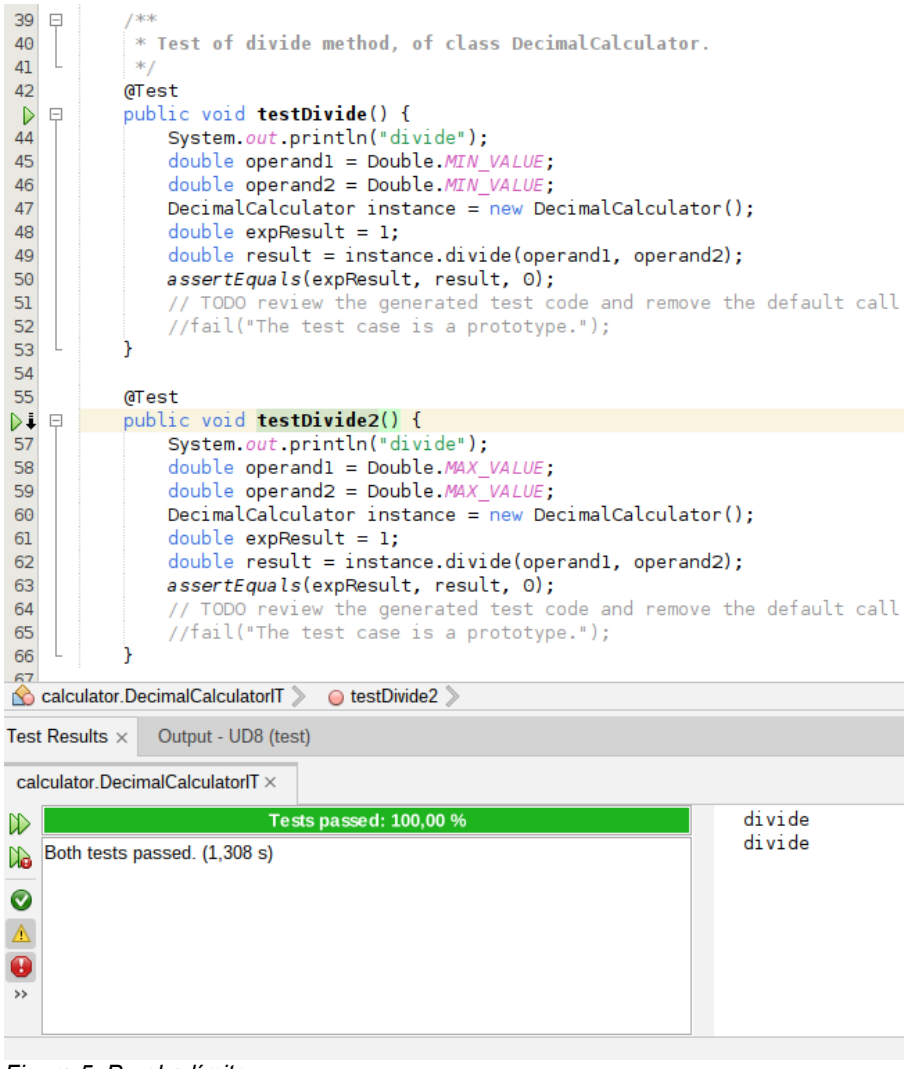


*Figura 5: Prueba límite*

## 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:

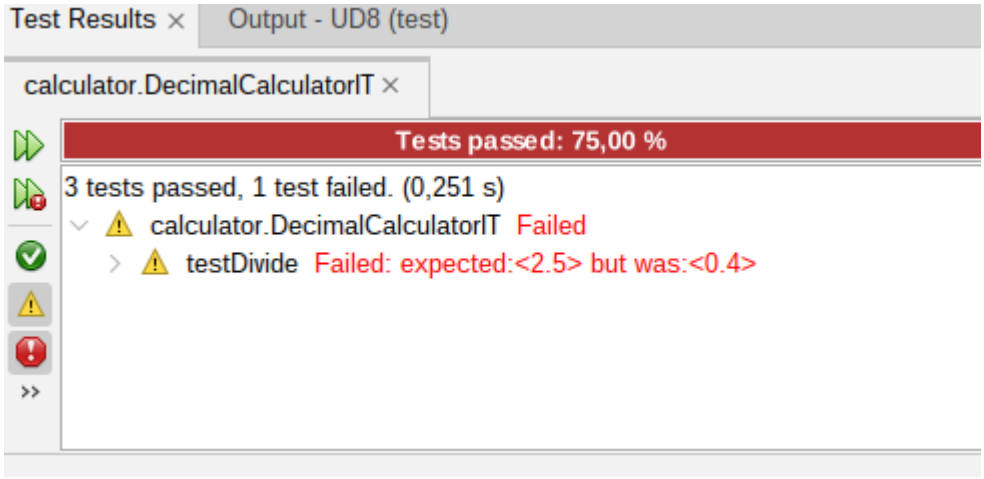


*Figura 6: Pruebas falladas*

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.

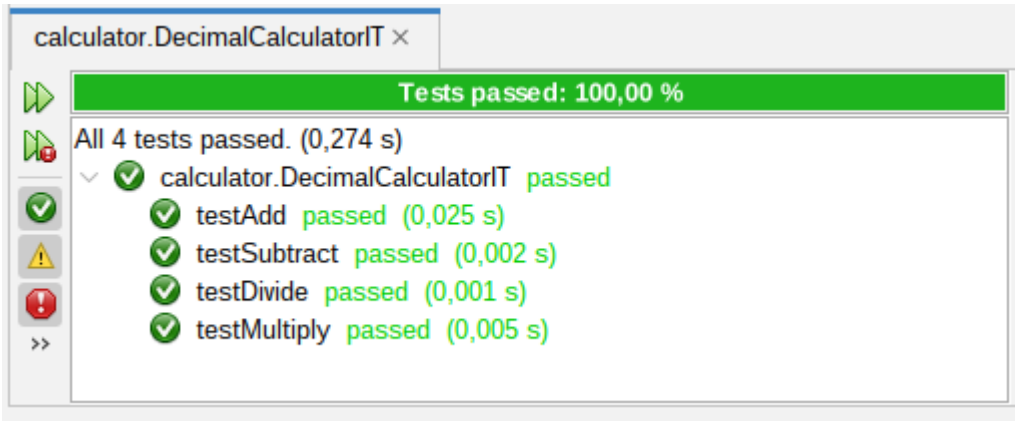


*Figura 7: Prueba de regresión superada*

## 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.

```java
        return operand1 * operand2;
    }

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

    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");
    }
}
```

Test Results | Output - UD8 (run) ×

```
run:
DC
Done: 11ms
BUILD SUCCESSFUL (total time: 0 seconds)
```

*Figura 8: Sobrecarga*

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.