



---

# TAREA DI08

---

Módulo de Desarrollo de Interfaces en modalidad a distancia del I.E.S.  
Augusto González de Linares.



15 DE FEBRERO DE 2023  
DIEGO GONZÁLEZ GARCÍA

# Índice

1. Ejercicio 1. ....	3
2. Ejercicio 2. ....	9
3. Ejercicio 3. ....	11
4. Ejercicio 4. ....	15
5. Ejercicio 5. ....	22
a. Pruebas de integración. ....	22
b. Pruebas de sistema. ....	25
c. Pruebas de regresión. ....	28
6. Ejercicio 6. ....	29
a. Pruebas funcionales ....	29
b. Pruebas de capacidad y rendimiento ....	29
c. Pruebas de uso de recursos ....	30
d. Pruebas de seguridad ....	30
e. Pruebas manuales y automáticas.....	31
f. Pruebas de usuarios ....	31
g. Pruebas de aceptación ....	32
7. Ejercicio 7. ....	33

# Enunciado.

En BK han recibido algunas quejas de clientes sobre defectos en su software.

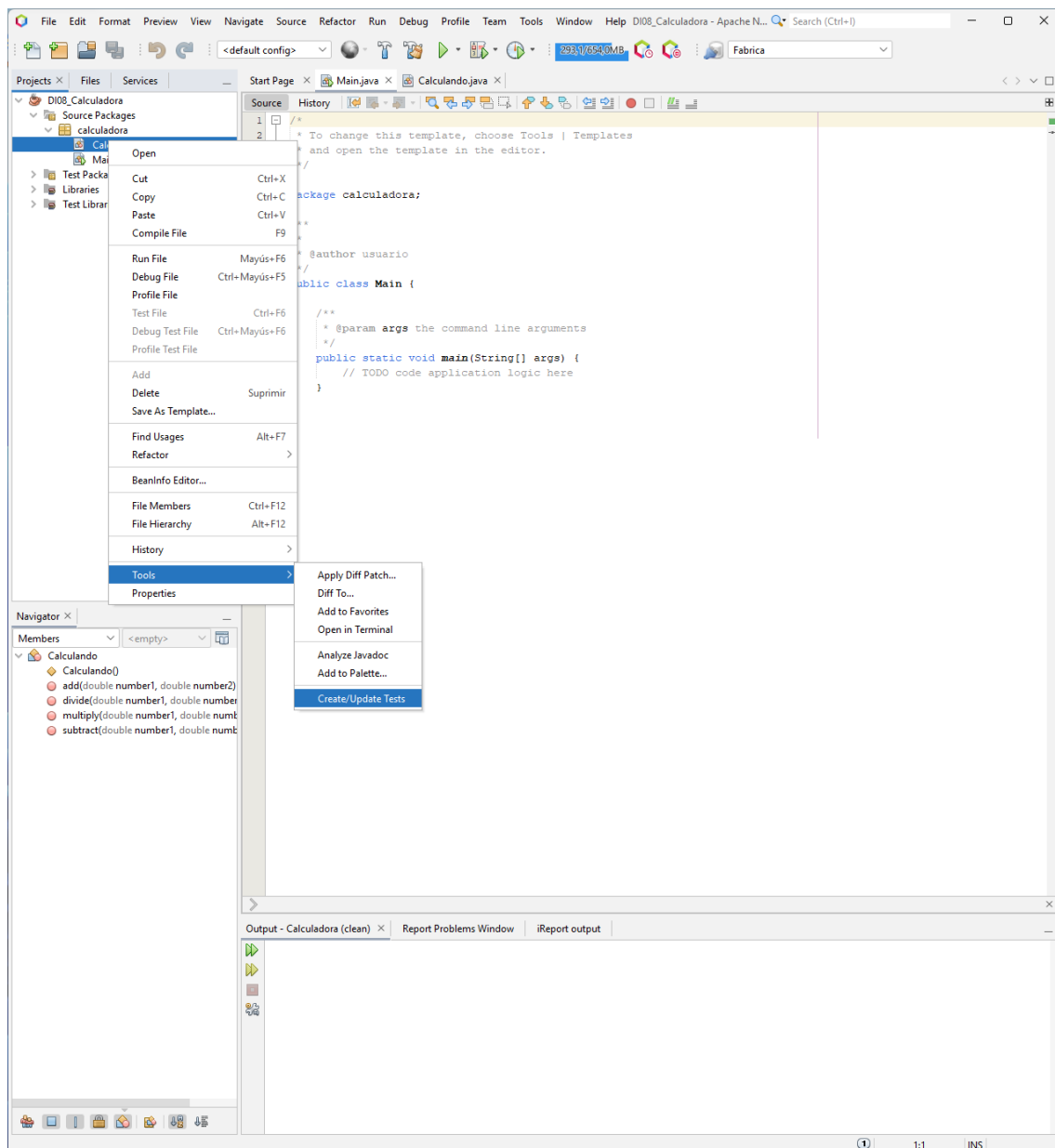
Ada está muy enfadada porque no se han seguido los protocolos de pruebas que la empresa tiene estandarizados. Por eso, en el nuevo proyecto que se va a desarrollar, tendrás que plantear la estrategia que asegure que los errores van a ser los mínimos posibles. Sabiendo que:

- Se trata de una aplicación desarrollada en Java
- Se van a realizar todas las pruebas vistas en la unidad.
- En principio, sólo se hará una versión por cada prueba.

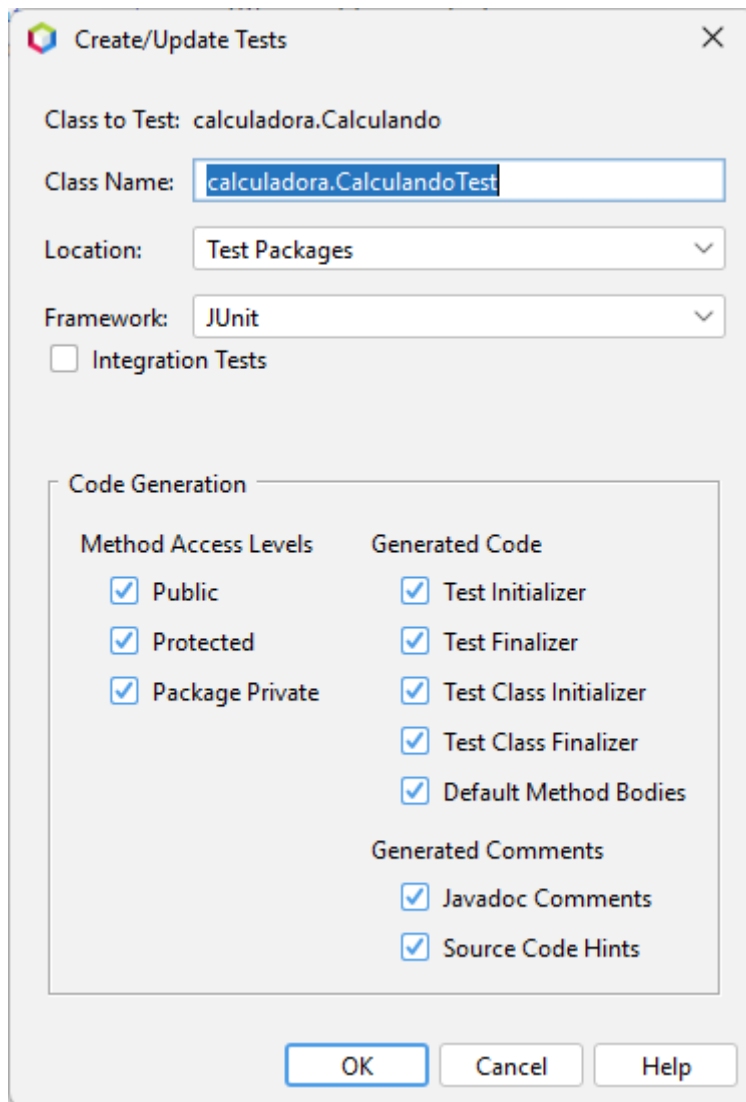
Para desarrollar esta actividad necesitarás tener instalado NetBeans y JUnit. Durante el desarrollo del módulo, hemos estado trabajando con la versión 8.2 de NetBeans, el cual ya trae incorporado Junit. En el caso de utilizar otra versión, asegúrate de tener instalado Junit en NetBeans.

## 1. Ejercicio 1.

Descarga el proyecto Java y ábrelo con NetBeans. Observa los métodos definidos en la clase Calculando.java. Vamos a probar cada método de la clase con JUnit. Para ello, deberás de seleccionar la clase y en el menú Herramientas deberás de seleccionar la opción Create /update Tests. Nos aparecerá una ventana donde consta la clase a la que se le van a realizar las pruebas y la ubicación de las mismas. Seleccionaremos como Framework Junit y veremos que el código de la aplicación importa automáticamente el framework Junit. Como solución a este apartado deberás de aportar el código de la clase generado. (Calificación 1 punto)



Procedemos a ejecutar el asistente para crear los test, pulsando con el botón derecho sobre la clase que queremos testear, seleccionamos *Tools* y a continuación *Create/Update Test*.



En el asistente nos aseguramos del nombre que va a tener la clase test, donde se va a guardar y que el framework utilizado es *JUnit*. Además, se nos ofrecen diferentes opciones para la generación del código.

Tras pulsar *OK*, nos genera el siguiente código:

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
 * default.txt to change this license
 * Click
 * nbfs://nbhost/SystemFileSystem/Templates/UnitTests/JUnit3TestClass.java
 * to edit this template
 */
package calculadora;

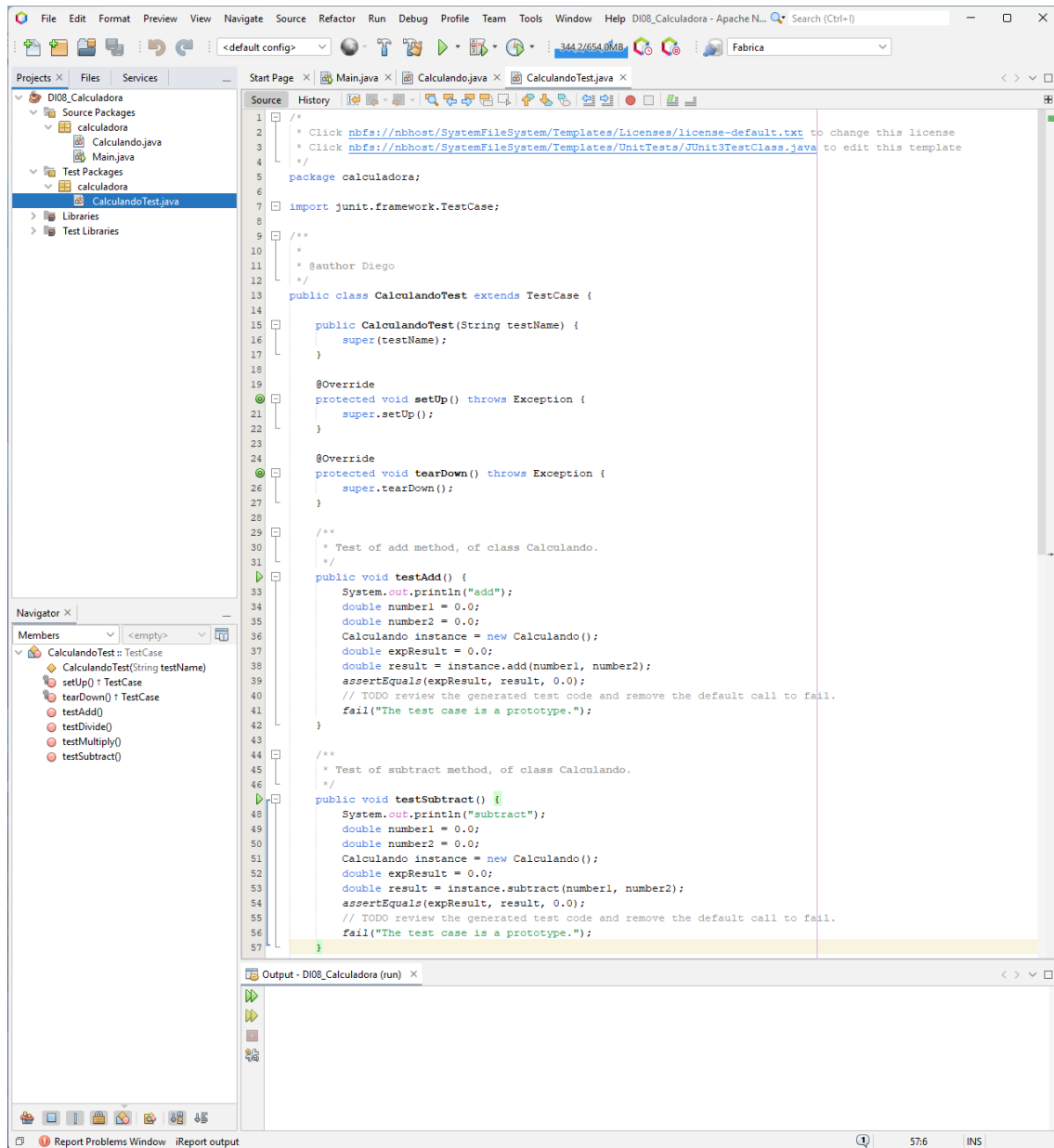
import junit.framework.TestCase;

/**
 *
 * @author Diego
```

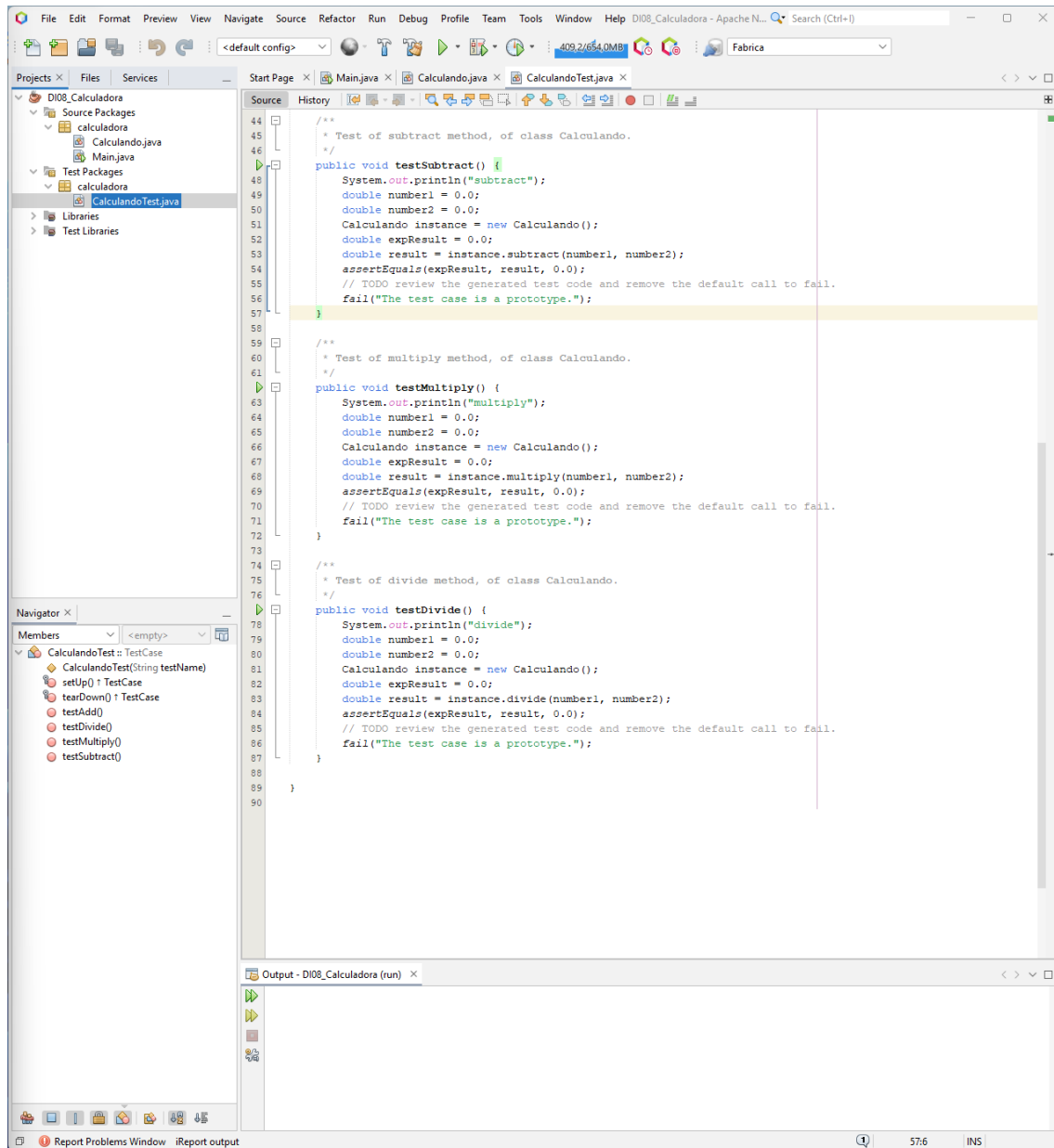
```
*/  
public class CalculandoTest extends TestCase {  
  
    public CalculandoTest(String testName) {  
        super(testName);  
    }  
  
    @Override  
    protected void setUp() throws Exception {  
        super.setUp();  
    }  
  
    @Override  
    protected void tearDown() throws Exception {  
        super.tearDown();  
    }  
  
    /**  
     * Test of add method, of class Calculando.  
     */  
    public void testAdd() {  
        System.out.println("add");  
        double number1 = 0.0;  
        double number2 = 0.0;  
        Calculando instance = new Calculando();  
        double expectedResult = 0.0;  
        double result = instance.add(number1, number2);  
        assertEquals(expectedResult, result, 0.0);  
        // TODO review the generated test code and remove the default  
call to fail.  
        fail("The test case is a prototype.");  
    }  
  
    /**  
     * Test of subtract method, of class Calculando.  
     */  
    public void testSubtract() {  
        System.out.println("subtract");  
        double number1 = 0.0;  
        double number2 = 0.0;  
        Calculando instance = new Calculando();  
        double expectedResult = 0.0;  
        double result = instance.subtract(number1, number2);  
        assertEquals(expectedResult, result, 0.0);  
        // TODO review the generated test code and remove the default  
call to fail.  
        fail("The test case is a prototype.");  
    }  
}
```

```
/**
 * Test of multiply method, of class Calculando.
 */
public void testMultiply() {
    System.out.println("multiply");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expResult, result, 0.0);
    // TODO review the generated test code and remove the default
call to fail.
    fail("The test case is a prototype.");
}

/**
 * Test of divide method, of class Calculando.
 */
public void testDivide() {
    System.out.println("divide");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, 0.0);
    // TODO review the generated test code and remove the default
call to fail.
    fail("The test case is a prototype.");
}
}
```

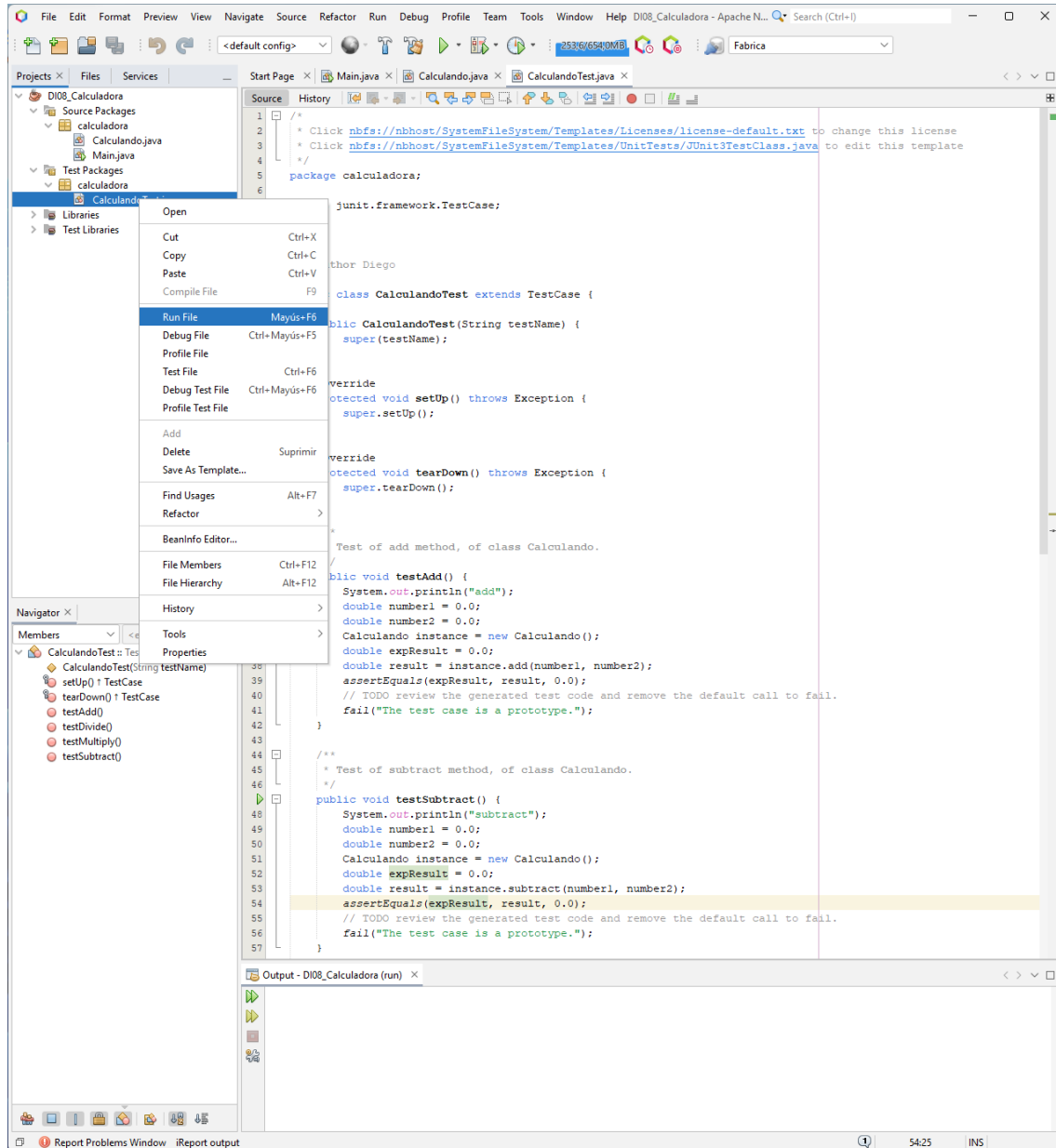




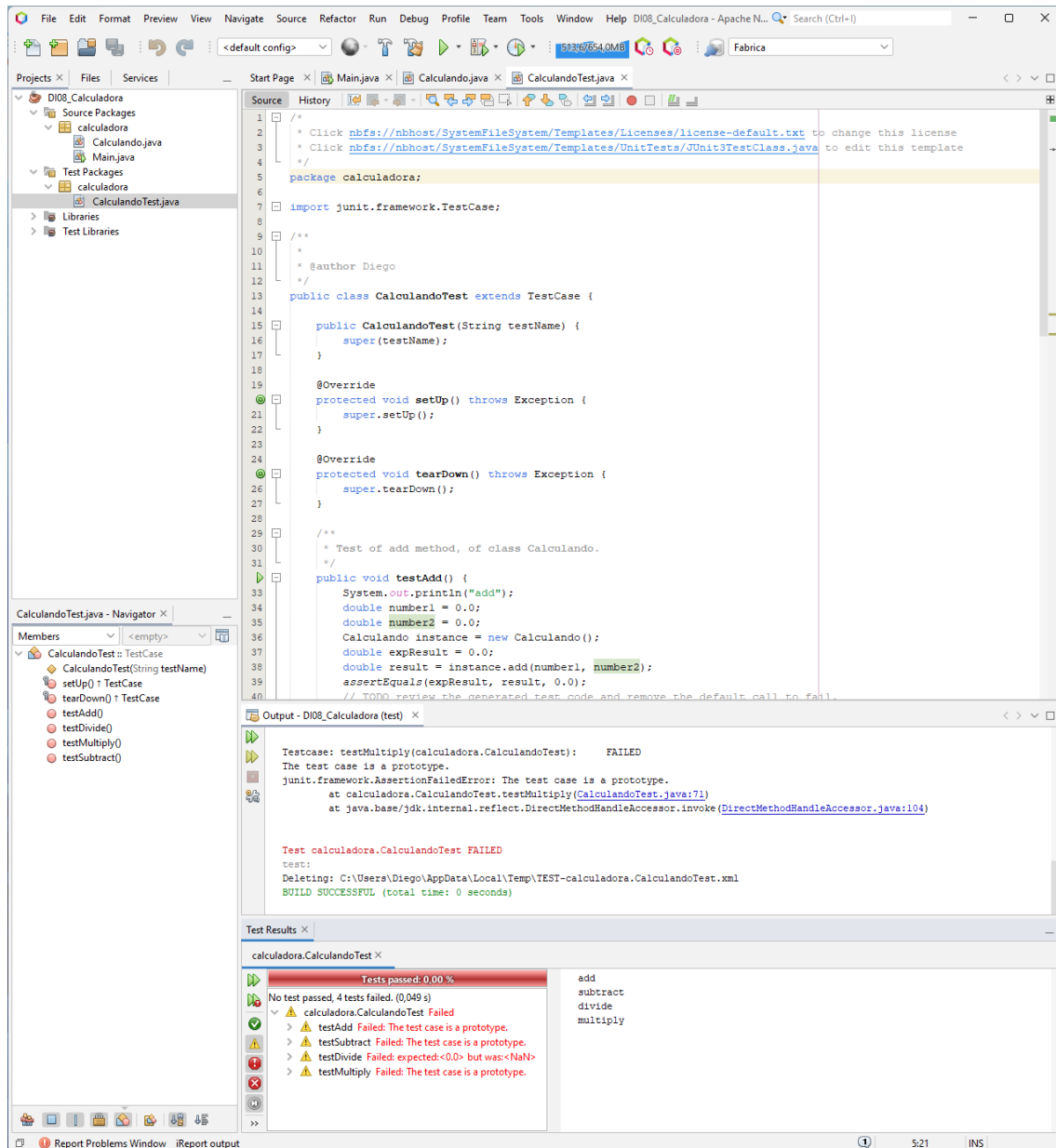


## 2. Ejercicio 2.

Selecciona la nueva clase de pruebas que has generado. Ejecútala. Realiza una captura de la ventana Test results como solución a este apartado. (Calificación 1 punto)



Pulsando con el botón derecho sobre la clase de test, seleccionamos la opción *Run File*.



Como podemos observar, la prueba ha pasado en un 0%, fallando los 4 test que se han realizado.

### 3. Ejercicio 3.

Accede al código de la clase de pruebas y elimina las líneas:

```
// TODO review the generated test code and remove the default call
to fail.
fail("The test case is a prototype.");
```

que aparece al final de cada método. Como solución a este apartado deberás de entregar el código de la clase generado. (Calificación 1 punto)

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
default.txt to change this license
 * Click
nbfs://nbhost/SystemFileSystem/Templates/UnitTests/JUnit3TestClass.java
to edit this template
 */
package calculadora;

import junit.framework.TestCase;

/**
 *
 * @author Diego
 */
public class CalculandoTest extends TestCase {

    public CalculandoTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

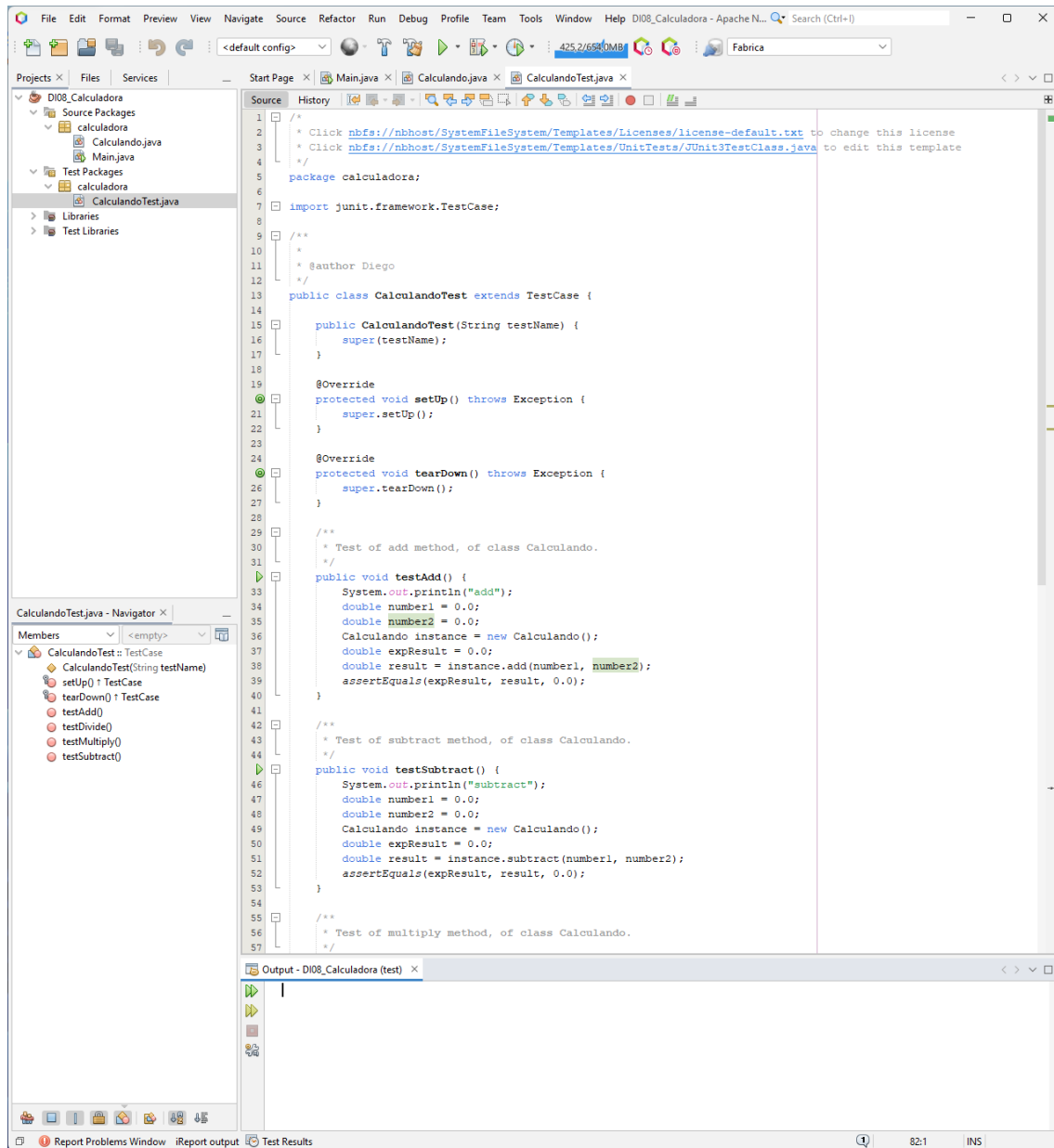
    /**
     * Test of add method, of class Calculando.
     */
    public void testAdd() {
        System.out.println("add");
        double number1 = 0.0;
        double number2 = 0.0;
```

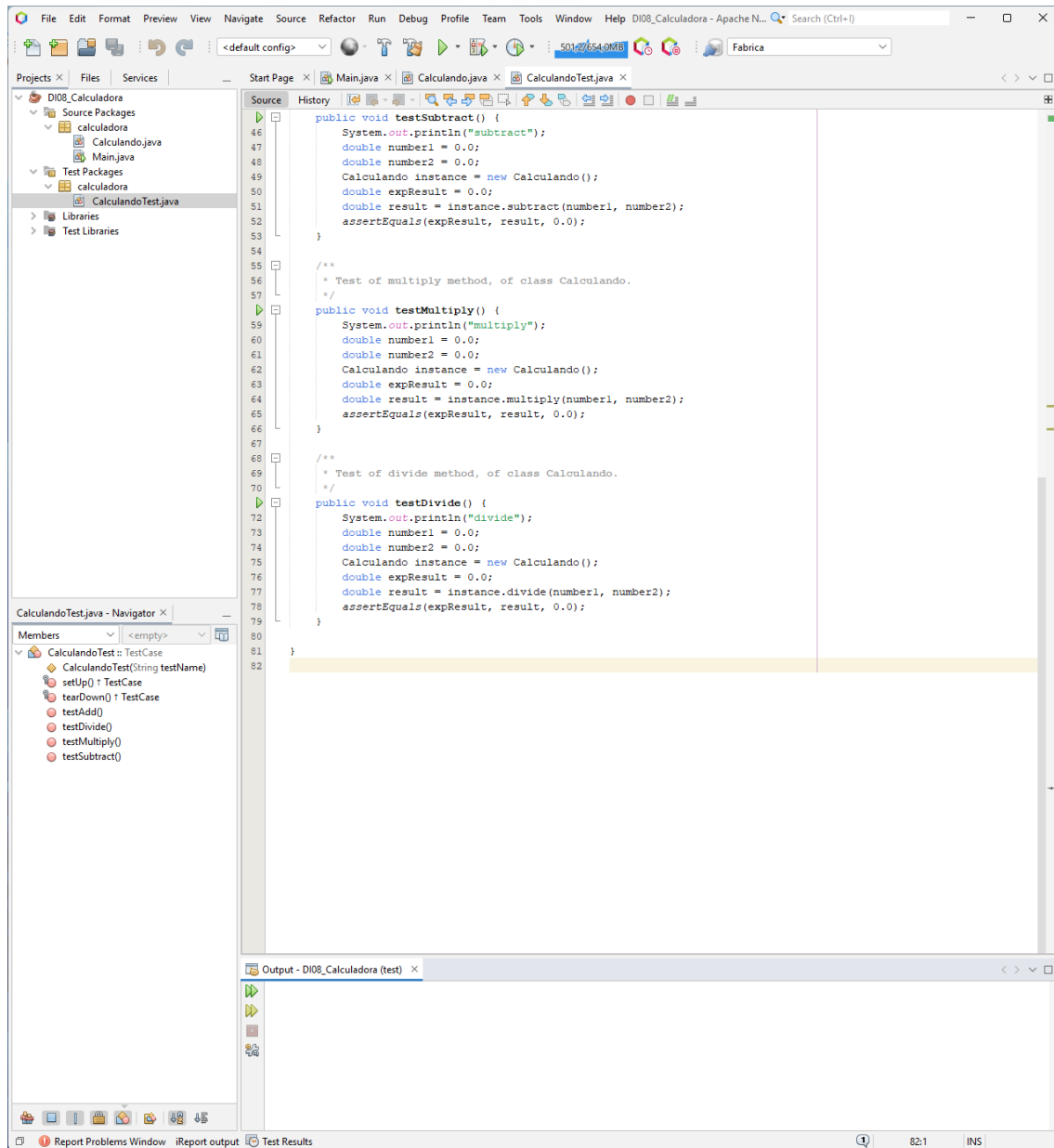
```
        Calculando instance = new Calculando();
        double expResult = 0.0;
        double result = instance.add(number1, number2);
        assertEquals(expResult, result, 0.0);
    }

    /**
     * Test of subtract method, of class Calculando.
     */
    public void testSubtract() {
        System.out.println("subtract");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expResult = 0.0;
        double result = instance.subtract(number1, number2);
        assertEquals(expResult, result, 0.0);
    }

    /**
     * Test of multiply method, of class Calculando.
     */
    public void testMultiply() {
        System.out.println("multiply");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expResult = 0.0;
        double result = instance.multiply(number1, number2);
        assertEquals(expResult, result, 0.0);
    }

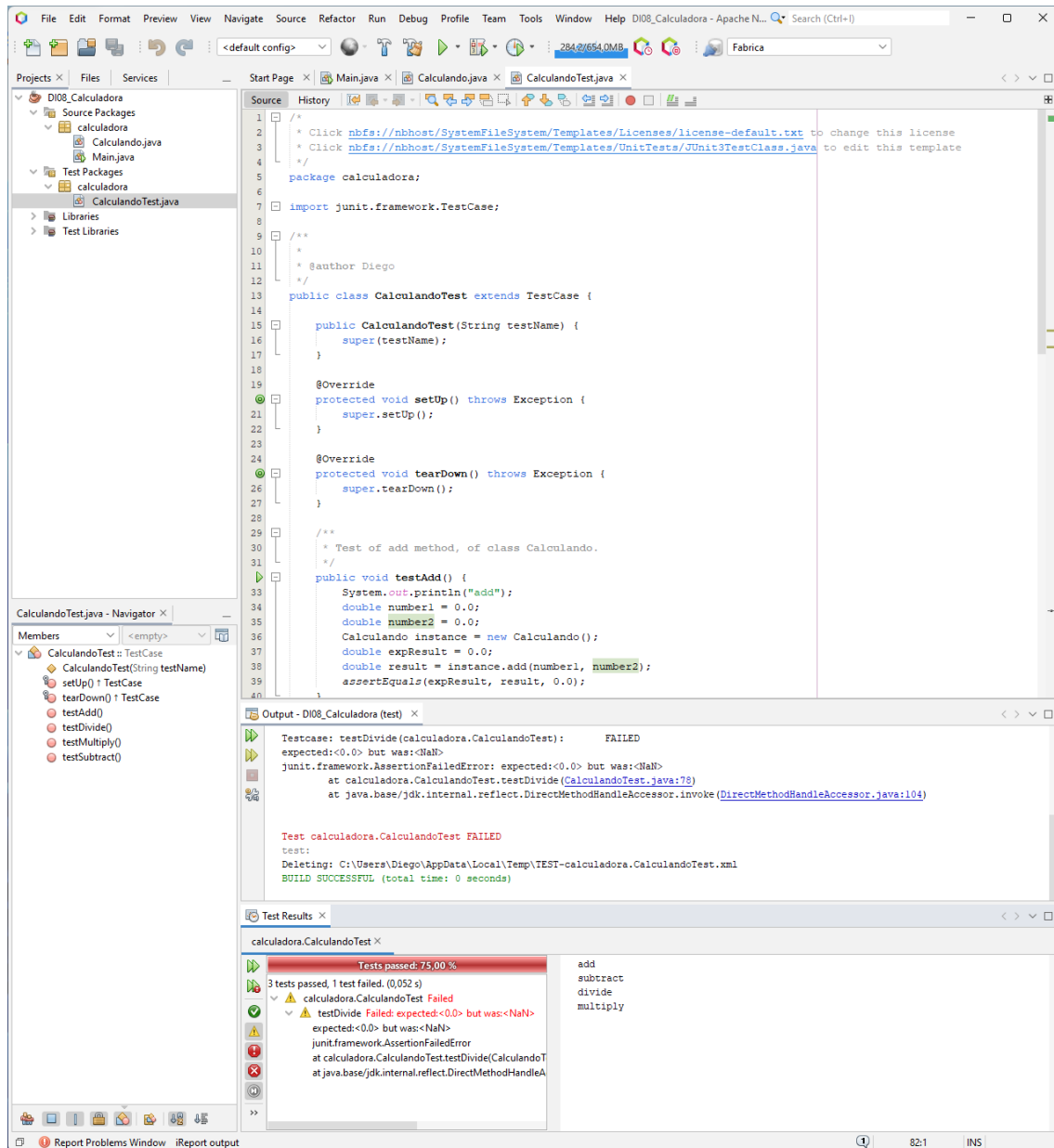
    /**
     * Test of divide method, of class Calculando.
     */
    public void testDivide() {
        System.out.println("divide");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expResult = 0.0;
        double result = instance.divide(number1, number2);
        assertEquals(expResult, result, 0.0);
    }
}
```





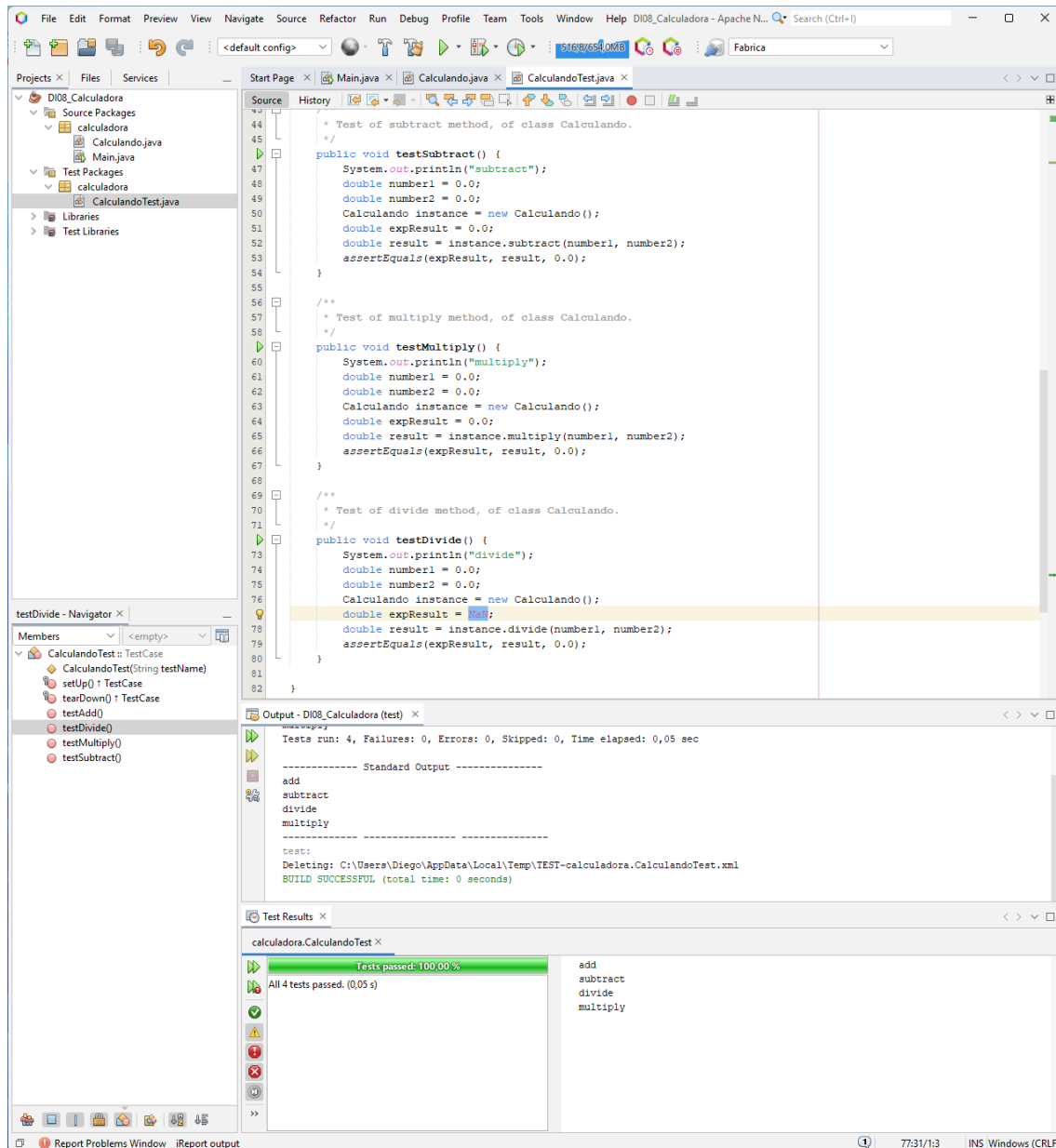
## 4. Ejercicio 4.

Selecciona la clase de prueba y ejecútala de nuevo. Debes de corregir todos los errores asignándole valores a las variables. Al final, debes de conseguir que la ejecución de la prueba sea satisfactoria. Como solución a este apartado deberás de aportar el código de la clase de prueba una vez que ha sido modificado para conseguir que las pruebas fueran satisfactorias. (Calificación 1 punto)



Como podemos observar, la prueba nos da de nuevo fallo, pero esta vez solo en el test de la división. Nos indica que el valor esperado es un *0.0*, pero su resultado ha sido *NaN* (NetBeans identifica de esta forma el valor infinito).





Como la división de  $0 / 0 = \text{infinito}$ , corregimos nuestro código añadiendo como valor esperado *NaN* (necesita de la importación de la librería *java.lang.Double.NaN*) y vemos que ya nos pasa todos los test satisfactoriamente.

```

/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
 * default.txt to change this license
 * Click
 * nbfs://nbhost/SystemFileSystem/Templates/UnitTests/JUnit3TestClass.java
 * to edit this template
 */
package calculadora;

import static java.lang.Double.NaN;
import junit.framework.TestCase;

```

```
/**
 *
 * @author Diego
 */
public class CalculandoTest extends TestCase {

    public CalculandoTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /**
     * Test of add method, of class Calculando.
     */
    public void testAdd() {
        System.out.println("add");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expectedResult = 0.0;
        double result = instance.add(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }

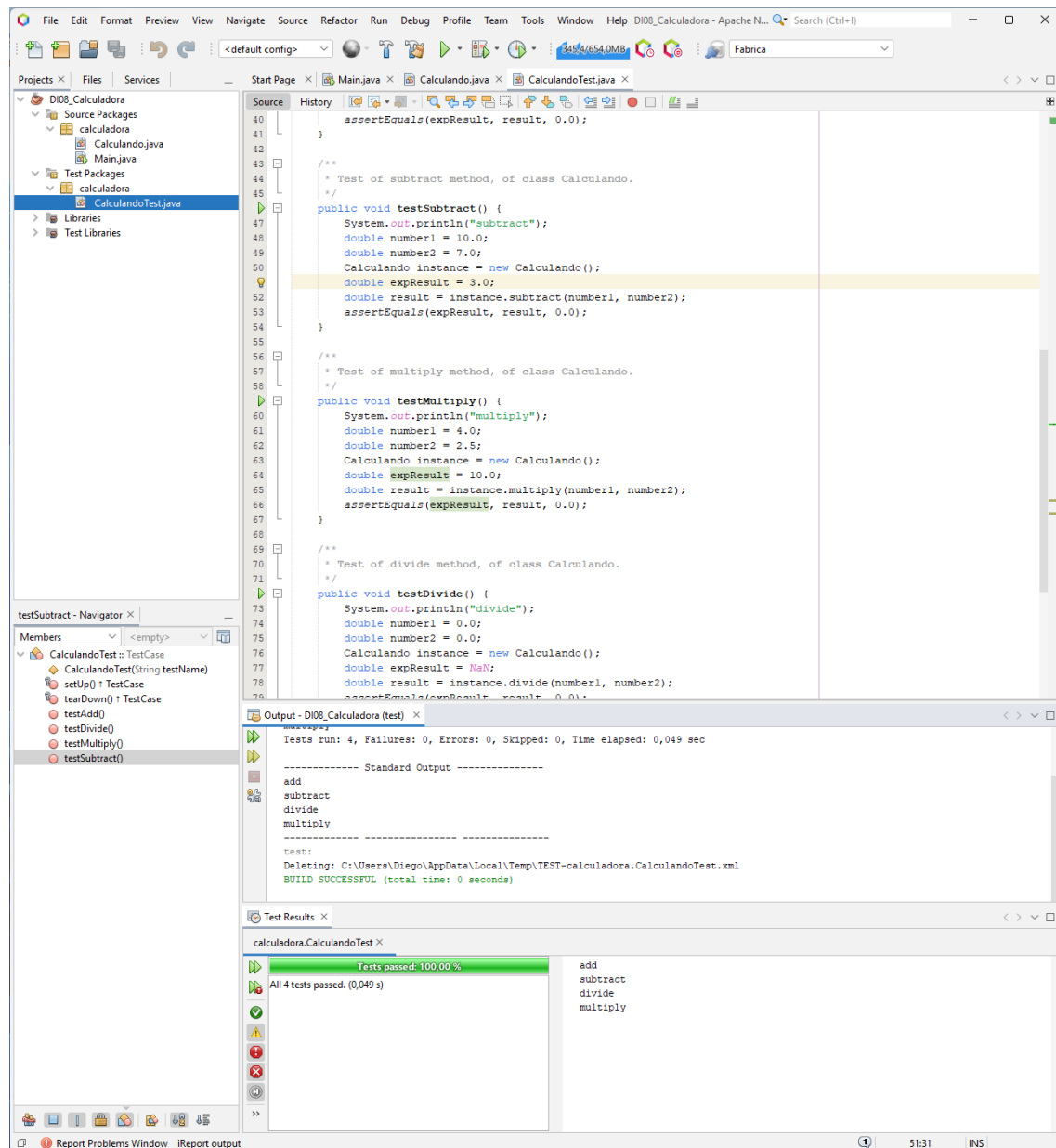
    /**
     * Test of subtract method, of class Calculando.
     */
    public void testSubtract() {
        System.out.println("subtract");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expectedResult = 0.0;
        double result = instance.subtract(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }

    /**
     * Test of multiply method, of class Calculando.
     */
}
```

```
public void testMultiply() {
    System.out.println("multiply");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result, 0.0);
}

/**
 * Test of divide method, of class Calculando.
 */
public void testDivide() {
    System.out.println("divide");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expectedResult = NaN;
    double result = instance.divide(number1, number2);
    assertEquals(expectedResult, result, 0.0);
}
}
```

Aunque las pruebas ya son satisfactorias, como todos los valores comprobados y esperados son 0, vamos a modificar las variables de los demás métodos y a realizar otra prueba.



La prueba de nuevo es satisfactoria tras modificar los valores comprobamos y esperados.

```

/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-
 default.txt to change this license
 * Click
 nbfs://nbhost/SystemFileSystem/Templates/UnitTests/JUnit3TestClass.java
 to edit this template
 */
package calculadora;

import static java.lang.Double.NaN;
import junit.framework.TestCase;

/**
 *
 */

```

```
* @author Diego
*/
public class CalculandoTest extends TestCase {

    public CalculandoTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /**
     * Test of add method, of class Calculando.
     */
    public void testAdd() {
        System.out.println("add");
        double number1 = 5.0;
        double number2 = 0.5;
        Calculando instance = new Calculando();
        double expectedResult = 5.5;
        double result = instance.add(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }

    /**
     * Test of subtract method, of class Calculando.
     */
    public void testSubtract() {
        System.out.println("subtract");
        double number1 = 10.0;
        double number2 = 7.0;
        Calculando instance = new Calculando();
        double expectedResult = 3.0;
        double result = instance.subtract(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }

    /**
     * Test of multiply method, of class Calculando.
     */
    public void testMultiply() {
        System.out.println("multiply");
    }
}
```

```
        double number1 = 4.0;
        double number2 = 2.5;
        Calculando instance = new Calculando();
        double expectedResult = 10.0;
        double result = instance.multiply(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }

    /**
     * Test of divide method, of class Calculando.
     */
    public void testDivide() {
        System.out.println("divide");
        double number1 = 0.0;
        double number2 = 0.0;
        Calculando instance = new Calculando();
        double expectedResult = NaN;
        double result = instance.divide(number1, number2);
        assertEquals(expectedResult, result, 0.0);
    }
}
```

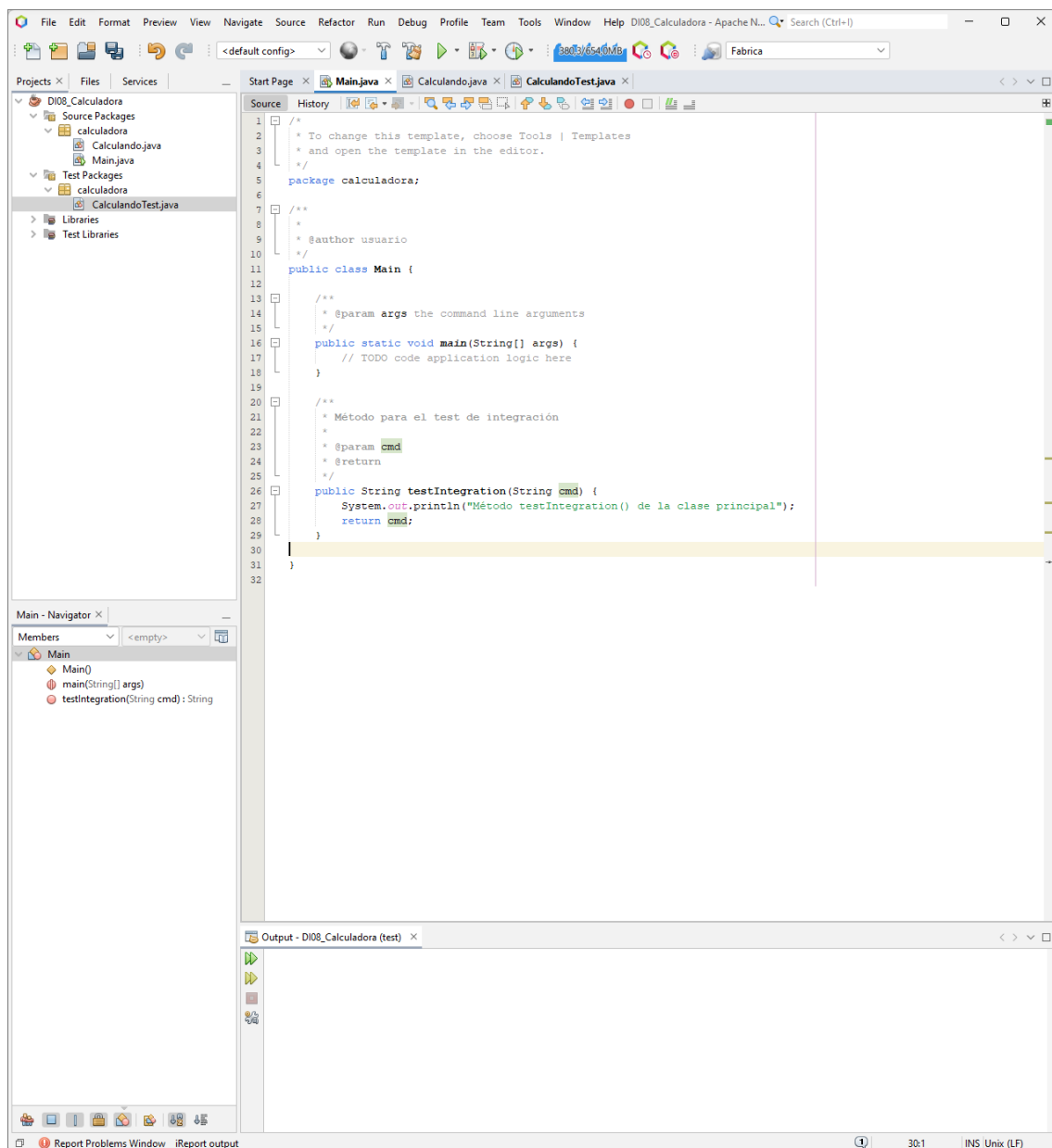
## 5. Ejercicio 5.

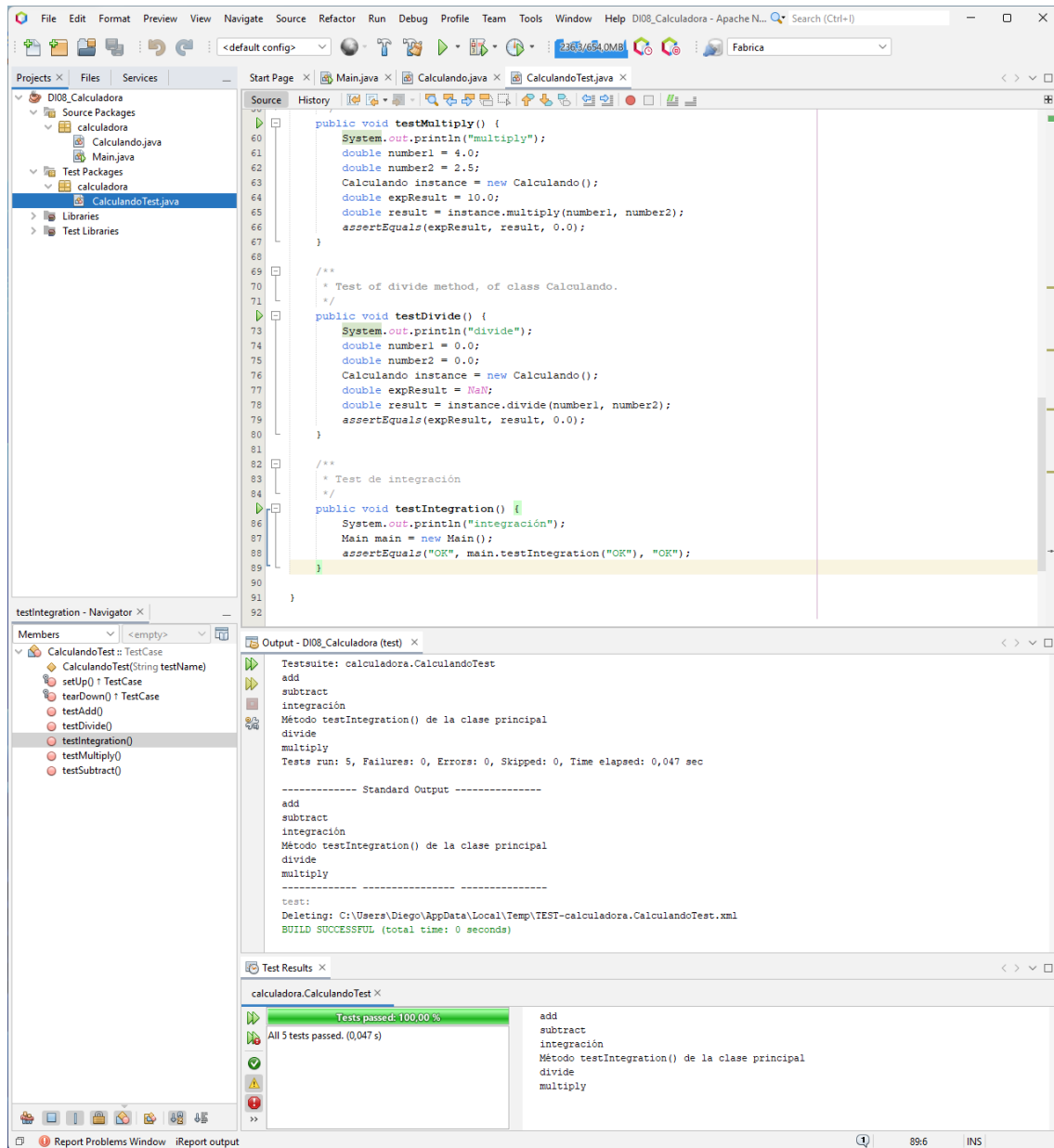
Implementa la planificación de las pruebas de integración, sistema y regresión. (Calificación 2 puntos)

### a. Pruebas de integración.

Las pruebas de integración consisten en verificar que el software, en conjunto, cumple su misión y se realiza sobre la unión de todos los módulos a la vez, para probar que la interrelación entre ellos no da lugar a ningún error o defecto. Son muy importantes por el hecho de que los módulos que componen una aplicación suelen fallar cuando trabajan de forma conjunta, aunque previamente se haya demostrado que trabajan bien de manera individual.

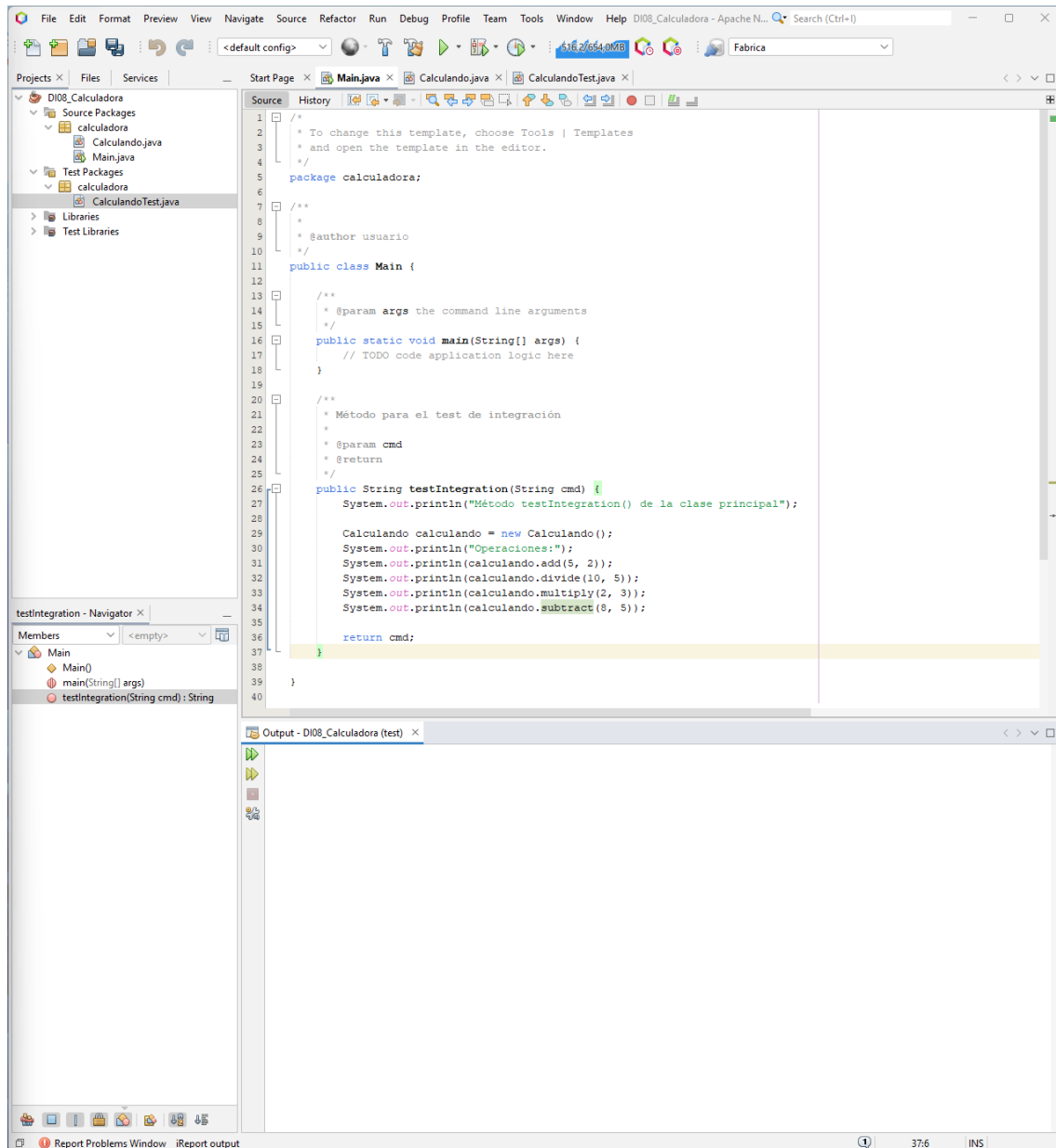
Para realizar estas pruebas, en primer lugar, vamos a crear un método en la clase principal llamado *testIntegration*, al cual llamaremos desde la clase de test, instanciando el método *main*.



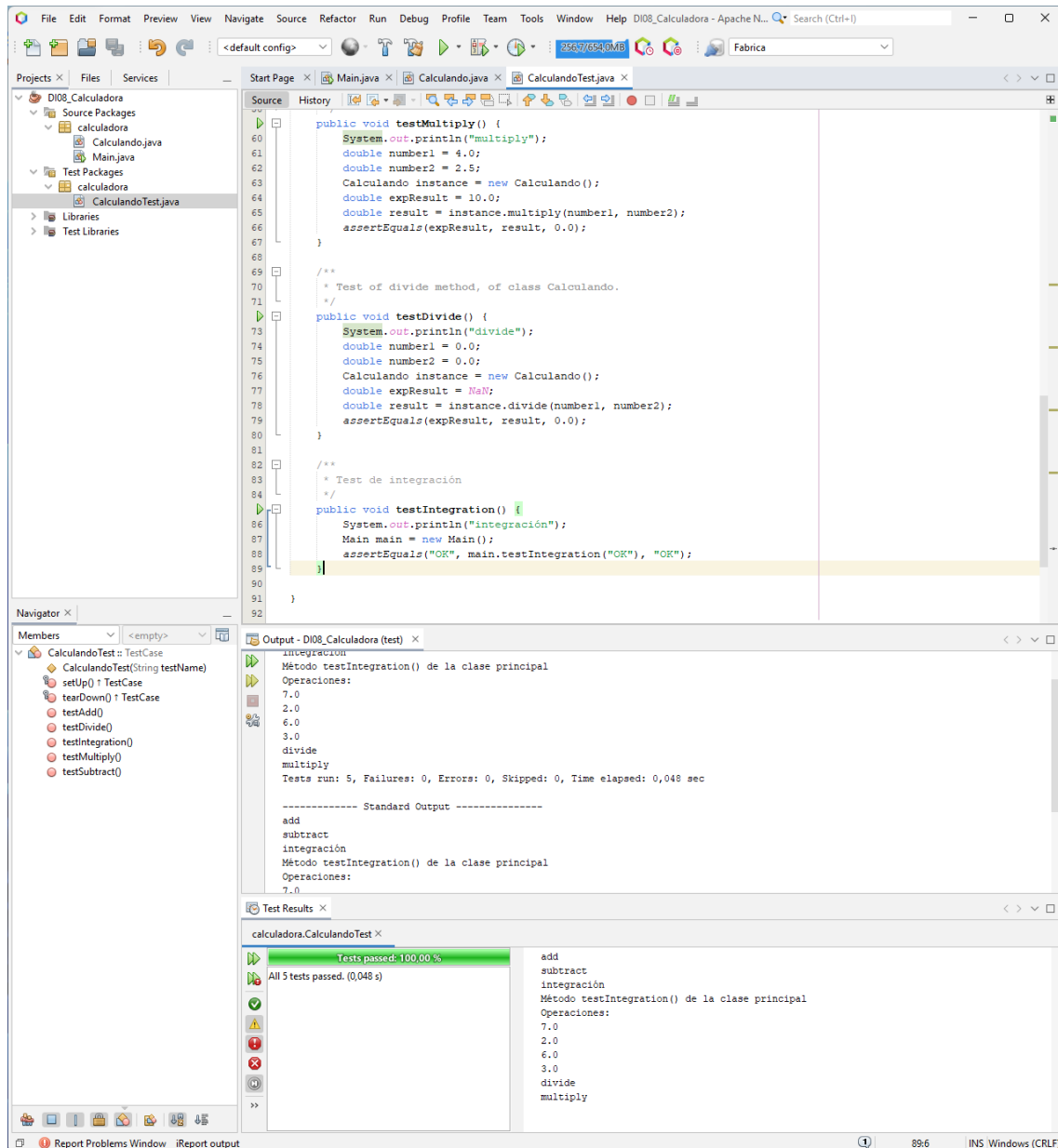


Desde la clase de test, una vez configurado el nuevo método para la integración, ejecutamos las pruebas que se pasan satisfactoriamente.





Ahora lo que hacemos es modificar el método `testIntegration` de la clase principal, en el cual instanciamos un objeto `Calculando` y utilizamos sus métodos. De esta manera verificamos que el conjunto de módulos cumple con su misión sin dar lugar a algún error de interrelación.



Como podemos observar el test pasa satisfactoriamente, en consecuencia, se ha superado la prueba de integración.

## b. Pruebas de sistema.

Las pruebas de sistema verifican el comportamiento del sistema en su conjunto. En concreto, se comprueban los requisitos no funcionales de la aplicación: Seguridad, velocidad, exactitud y fiabilidad. También se prueban los interfaces externos con otros sistemas, utilidades, unidades físicas y el entorno operativo.

Entre las pruebas de sistema más relevantes encontramos las de configuración y recuperación.

### ➤ Pruebas de configuración.

Estas pruebas verifican la instalación del software en el entorno de destino y comprueban el comportamiento del sistema frente a los requisitos de configuración. Además, analizan el software bajo configuraciones diferentes para diferentes usuarios.

La meta de esta prueba es hacer que la aplicación falle en el cumplimiento de los requerimientos de configuración, de manera que los defectos escondidos sean identificados, analizados, arreglados y prevenidos en el futuro.

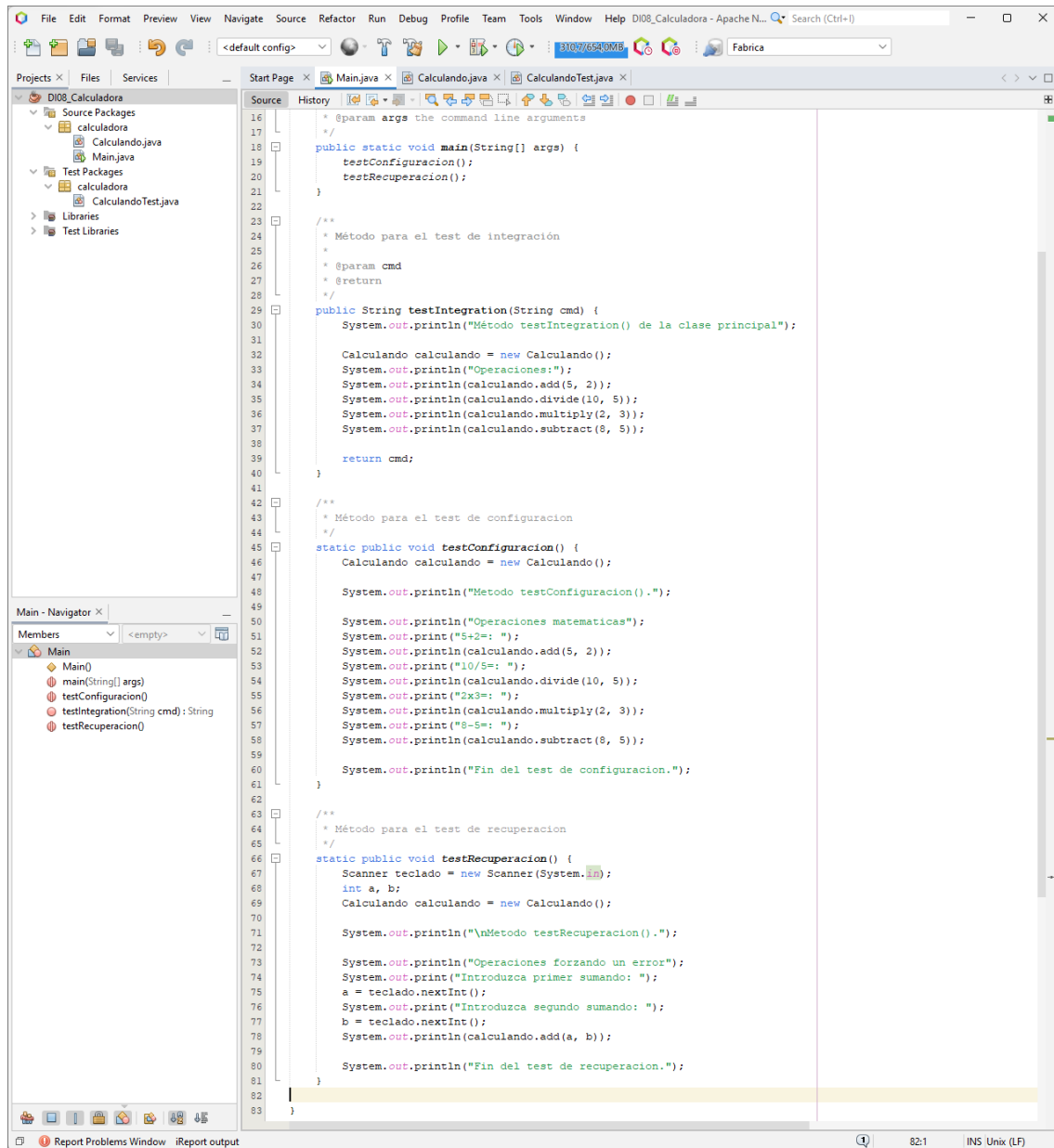
➤ **Pruebas de recuperación.**

La prueba de recuperación es una prueba de sistema que fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo de forma satisfactoria.

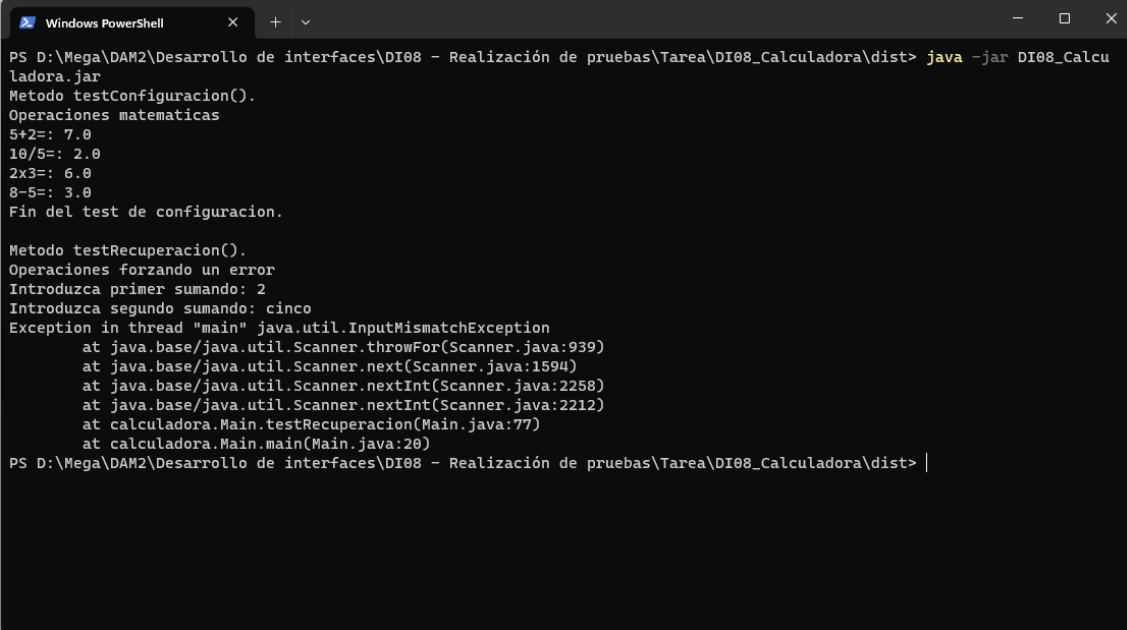
Es un parámetro muy importante de la aplicación, porque si un sistema no es capaz de recuperarse ante una entrada no válida o de un fallo súbito, la calidad de nuestro software quedará en entredicho.

Para realizar las pruebas de sistema, en primer lugar vamos a crear dos métodos en la clase principal llamados *testSistema* y *testRecuperacion*, a los cuales llamaremos desde el método *main*.

- El *testConfiguracion*, consiste en la impresión por pantalla de diferentes operaciones matemáticas.
- El *testRecuperacion*, consiste en la solicitud de dos sumandos por teclado para realizar la operación matemática, pero en uno de ellos en lugar de introducir un número, introduciremos texto, forzando de esta forma un error y comprobar con esto si se recupera el programa.



Tras esto, crearemos el archivo `.jar` de la aplicación, para posteriormente ejecutarlos en nuestro sistema y realizar los test.



```
PS D:\Mega\DAM2\Desarrollo de interfaces\DI08 - Realización de pruebas\Tarea\DI08_Calculadora\dist> java -jar DI08_Calculadora.jar
Metodo testConfiguracion().
Operaciones matematicas
5+2=: 7.0
10/5=: 2.0
2x3=: 6.0
8-5=: 3.0
Fin del test de configuracion.

Metodo testRecuperacion().
Operaciones forzando un error
Introduzca primer sumando: 2
Introduzca segundo sumando: cinco
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at calculadora.Main.testRecuperacion(Main.java:77)
    at calculadora.Main.main(Main.java:20)
PS D:\Mega\DAM2\Desarrollo de interfaces\DI08 - Realización de pruebas\Tarea\DI08_Calculadora\dist> |
```

Como podemos observar, el test de configuración se ha pasado satisfactoriamente, pero el test de recuperación ha dado un error del que no se a recuperado, finalizando de forma inesperada la ejecución de la aplicación.

Como conclusión de las pruebas, entiendo que este caso de introducir un texto en lugar de un número se debería de controlar por código, ya sea mostrando un aviso y volviendo a solicitar un número o controlando la excepción para impedir el cierre súbito de la aplicación, aunque no se ejecute la operación solicitada.

### c. Pruebas de regresión.

Cada vez que se añade un nuevo módulo a la aplicación, o cuando se modifica, el software cambia. Como resultado de esa modificación (o adición) de módulos, podemos introducir errores en el programa que antes no teníamos. Por ello, se hace necesario volver a probar la aplicación.

La prueba de regresión consiste en volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados. Esta prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas (es lo más frecuente).

Para realizar esta prueba, lo primero que necesitaríamos sería añadir mas funciones nuevas a nuestra aplicación. Si suponemos que hemos añadido a nuestra aplicación una función para realizar por ejemplo raíces cuadradas, tras tener dicha opción programada y funcionando correctamente, tendríamos que comprobar de nuevo las pruebas que hemos realizado hasta este momento sobre el resto de funciones/módulos y comprobar que todos los test son superados satisfactoriamente.

Esto en realidad es totalmente desaconsejable, ya que supone una gran perdida de tiempo volver a realizar todas y cada una de las pruebas sobre cada componente. Actualmente las pruebas de regresión son una parte integral del método de desarrollo de software conocido como *Programación Extrema*. Para ello se utilizan herramientas que permiten detectar este tipo de errores de manera parcial o totalmente automatizada en cada una de las fases del desarrollo.

## 6. Ejercicio 6.

Planifica las restantes pruebas, estableciendo qué parámetros se van a analizar. (Calificación 2 puntos)

### a. Pruebas funcionales

Las pruebas funcionales validan si el comportamiento observado del software es conforme con sus especificaciones. Normalmente, los responsables de realizar estas pruebas son los analistas de la aplicación con apoyo de los usuarios finales. Su aprobación dará paso a la fase de producción propiamente dicha.

Son **pruebas de caja negra** y las personas encargadas de realizar las pruebas, no les importa cómo se generan las respuestas, sólo analizan las salidas de la aplicación frente a sus entradas. Estas pruebas intentan responder a las preguntas: "¿funciona esta utilidad de la aplicación?", "¿el usuario podrá hacer esto?".

Dentro de las pruebas funcionales, encontramos tres tipos:

- **Análisis de valores límite:** Como entradas, seleccionamos aquellos valores que están en el límite donde la aplicación es válida. (Si estos valores no dan problemas, el resto tampoco los dará).
- **Particiones equivalentes:** Como entradas, se seleccionará una muestra representativa de todas las posibles. El resultado será extrapolable al resto.
- **Pruebas aleatorias:** Como entradas, se selecciona una muestra de casos de prueba al azar. Se utiliza sólo en aplicaciones no interactivas.

Para realizar esta prueba, en primer lugar, vamos a utilizar la técnica de clases de equivalencia, dando un valor representativo de cada clase. Esto permite suponer que el resultado que se obtiene será el mismo que con cualquier otro valor de la clase.

En nuestro caso, vamos a realizar la primera prueba introduciendo dos números (2, 8) para cada operación, de los cuales se espera que la aplicación nos devuelva un resultado numérico en todos los casos. Para la segunda prueba, introduciremos una cadena de caracteres y un número (dos, 8) para cada operación, de los cuales se espera que la aplicación en todos los casos nos dé un mensaje de error o no nos permita introducir los caracteres.

En segundo lugar, vamos a utilizar la técnica de análisis de valores límite. Como en nuestro caso la aplicación realiza operaciones matemáticas de sumar, restas, multiplicar y dividir, vamos a realizar nuestra prueba con el valor 0, que siempre se suele considerar un valor límite debido a su particularidad. Para este valor, nos esperamos que el resultado de las operaciones siempre sea 0, a excepción de la división, que como ya hemos visto en el punto 4 de la tarea, su resultado es infinito.

### b. Pruebas de capacidad y rendimiento

- **Pruebas de capacidad.**

La prueba de capacidad ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales. Es decir, determina hasta dónde el sistema es capaz de soportar determinadas condiciones extremas. Para realizar esta prueba se recurre a los siguientes métodos: Aumentar la frecuencia de entradas para ver cómo responde la aplicación. Se ejecutan casos que requieran el máximo de memoria. Se buscan una cantidad

máxima de datos que residan en disco. Se realizan pruebas de sensibilidad, intentando descubrir combinaciones de datos que puedan producir inestabilidad en el sistema.

Para realizar esta prueba vamos a programar el método principal para que ejecute 500.000 veces seguidas el método *testIntegration* que hemos programado anteriormente. De esta forma aumentamos muy considerablemente la frecuencia de ejecución de operaciones por parte de la aplicación y comprobaremos si es capaz de soportarlo.

➤ **Pruebas de rendimiento.**

La prueba de rendimiento determina los tiempos de respuesta, el espacio que ocupan los módulos en disco y en memoria, el flujo de datos que genera a través de un canal de comunicaciones, etc. La prueba de rendimiento está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado. Se realiza durante todos los pasos del proceso de la prueba. Incluso a nivel de unidad, se debe asegurar el rendimiento de los módulos individuales a medida que se llevan a cabo las pruebas de caja blanca. Sin embargo, hasta que no están completamente integrados todos los elementos del sistema no se puede asegurar realmente el rendimiento del sistema.

Para realizar esta prueba, vamos a ejecutar de nuevo el ejemplo anterior y vamos a comprobar el tiempo que tarda en realizar su ejecución. NetBeans nos lo pone muy fácil, ya que, tras cada ejecución de programa, nos muestra el tiempo que ha estado en ejecución, por lo cual podríamos obtener este valor fácilmente. Para comprobar el uso de disco y de memoria, abriremos el *administrador de tareas*, ya que en él podemos observar el uso de los recursos del sistema durante la ejecución.

### c. Pruebas de uso de recursos

La evolución de los sistemas informáticos lleva incorporada una mayor complejidad en su diseño y en los recursos hardware y software que requiere. Las aplicaciones requieren cada vez mayor memoria RAM y espacio en disco para funcionar correctamente y nuestros equipos habitualmente tienen limitaciones en este sentido. Es por ello importante el uso eficaz que hace una aplicación de los recursos de que dispone.

La optimización es necesaria para lograr un uso eficiente de recursos. La optimización se debe hacer tanto en el código como en el uso del código. Todo ello para lograr la adaptación del software a la arquitectura de destino y así reducir los tiempos de ejecución de forma automática.

En cambio, en la actualidad la preocupación por la eficiencia en cuanto a recursos de la máquina ha dejado paso a la preocupación por la usabilidad de la aplicación. Esto es así porque en estos momentos contamos con equipos con varios gigas de memoria RAM y varios núcleos de procesador, con discos duros de hasta uno o varios terabytes de capacidad.

Esta prueba la podemos realizar de la misma forma que la anterior, ya que la forma más fácil de comprobar el uso de los recursos del sistema es desde el *administrador de tareas*. Comprobaremos el uso de recursos por parte del sistema en reposo y mientras ejecuta las operaciones, teniendo en cuenta que no es recomendable ir al límite del uso de los recursos.

### d. Pruebas de seguridad

La prueba de seguridad intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán de accesos no autorizados. Mide la habilidad de un sistema para soportar ataques contra su seguridad. El ataque se puede ejecutar mediante los programas, los datos o

los documentos de la aplicación. Es decir, determinan los niveles de permiso de usuarios, las operaciones de acceso al sistema y acceso a los datos.

En concreto, se intenta verificar que el sistema es robusto frente a problemas de seguridad, tales como:

- Intentar conseguir las claves de acceso de cualquier forma.
- Atacar con software a medida.
- Bloquear el sistema.
- Provocar errores del sistema, entrando durante su recuperación.

En nuestro caso, esta prueba no tiene mucho sentido ya que nuestra aplicación no tiene ningún control de acceso.

### **e. Pruebas manuales y automáticas**

Todos los tipos de pruebas que estamos viendo se ejecutan de manera regular con el objetivo de probar que la evolución del software desarrollado funciona perfectamente, tanto para las funcionalidades nuevas como para aquellas que ya existían. Una de las claves es que las pruebas sean automatizadas, porque si se ejecutaran manualmente, el coste de hacerlo "continuamente" sería inasumible.

Para evitar este coste, se deben de automatizar todo lo que sea posible las pruebas y así facilitarnos la vida. En el mercado disponemos de una amplia gama de herramientas en este sentido. Existen infinidad de herramientas gratuitas que, según las características de nuestro proyecto (número de requisitos, grado de estabilidad, organización del equipo, etc.), podemos valorar muy positivamente.

Según Meudec, las herramientas de pruebas de software persiguen automatizar tres tareas fundamentales:

- Tareas administrativas y generación de documentación.
- Tareas mecánicas (ejecución).
- Tareas de generación de casos de prueba.

### **f. Pruebas de usuarios**

La prueba de usuario consiste en asegurar que la interfaz de usuario de la aplicación sea amigable, intuitiva y que funcione correctamente. Se determina la calidad del software en la forma en la que el usuario interactúa con el sistema, considerando la facilidad de uso y satisfacción del cliente.

Es obvio que el estudio de la reacción de los individuos ante nuestra aplicación es un proceso complicado, pero existen varias métricas que implican un grado de fiabilidad que queda cubierto en un entorno de pruebas automatizadas.

Para verificar la facilidad de uso de una aplicación se pueden incluir en los ensayos de pruebas:

- Encuestas y entrevistas.
- Grupos de enfoque.
- Pruebas con herramientas de software avanzado.
- Evaluaciones realizadas por expertos.



### g. Pruebas de aceptación

La prueba de aceptación sirve para comprobar que el software cumple con los requerimientos iniciales y que funciona de acuerdo con lo que el cliente esperaba de él, es decir, satisfaciendo sus expectativas. Por eso, y para que la prueba sea objetiva, es el mismo cliente quien la realiza. Podemos escuchar comentarios como: "esto no era lo que yo quería" o "esto no es lo que yo necesito", e incluso "esto no es lo que yo pedí".

En la prueba de aceptación se evalúa:

- Que se satisfacen los requisitos funcionales.
- Que se alcanzan los requisitos de rendimiento.
- Que la documentación es correcta e inteligible.

## 7. Ejercicio 7.

Supuestas exitosas las pruebas, documenta el resultado (Calificación 2 puntos).

Este apartado lo he ido haciendo al mismo tiempo que realizaba las pruebas de cada apartado. Como se puede observar al final del apartado 4, 5a y 5b, apartado de pruebas de configuración, todas las pruebas finalizaron de forma exitosa y se muestran sus resultados.