Enunciado.

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

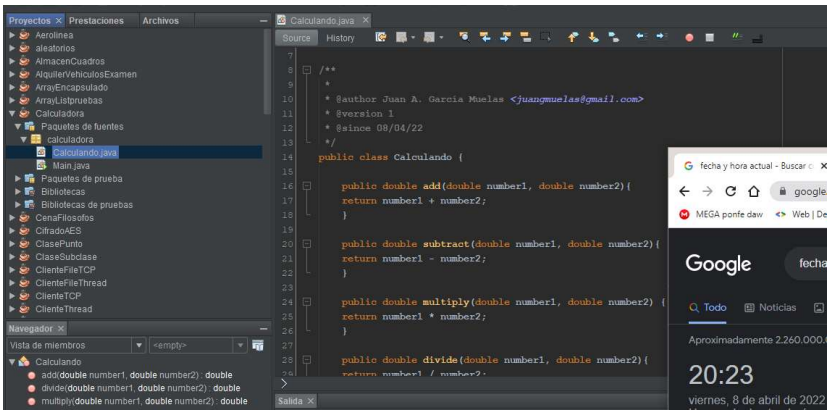
Ada está muy enfadada porque no se han seguido los protocolos de pruebas que la empresa tiene estandarizados. Por eso, en el nuevo proyecto que se va a desarrollar, tendrás que plantear la estrategia que asegure que los errores van a ser los mínimos posibles. Sabiendo que:

- ✓ 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.

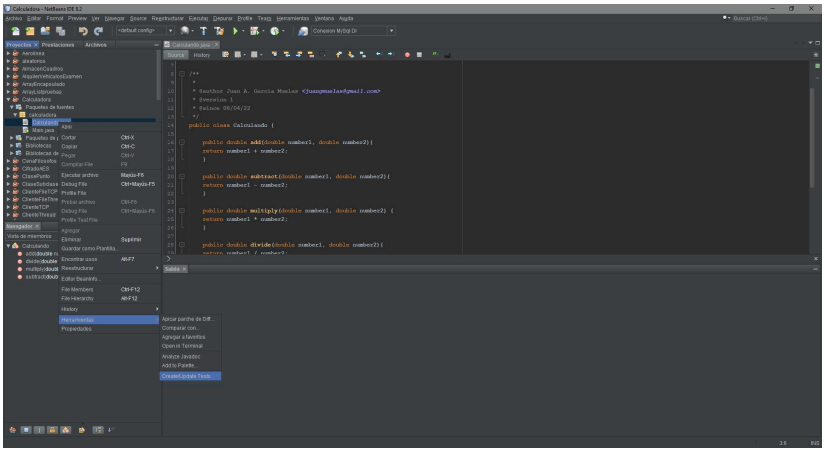
Para desarrollar esta actividad necesitarás tener instalado NetBeans y JUnit. Durante el desarrollo del módulo, hemos estado trabajando con la versión 8.2 de NetBeans, el cual ya trae incorporado Junit. En el caso de utilizar otra versión, asegúrate de tener instalado Junit en NetBeans.

1. Más abajo puedes descargar el Proyecto Java, 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 estas. 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.

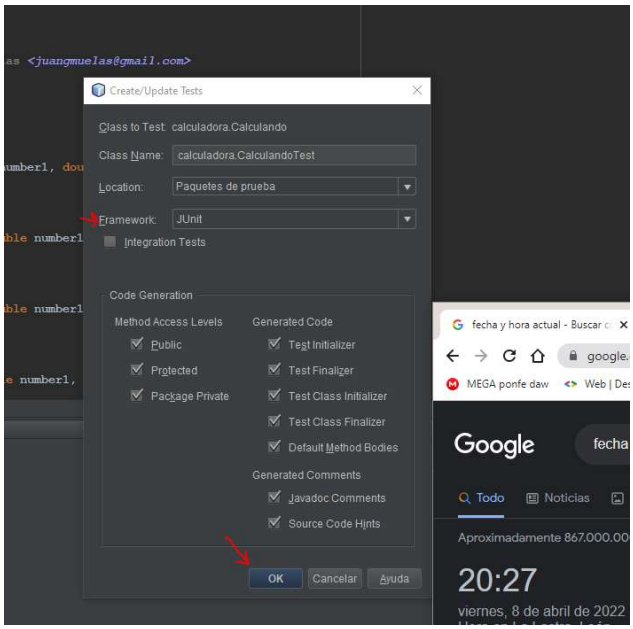
   Sigo los pasos pautados en el enunciado, y tras importar el proyecto `Calculadora`, genero el archivo de test.



Creación de proyecto Calculadora.



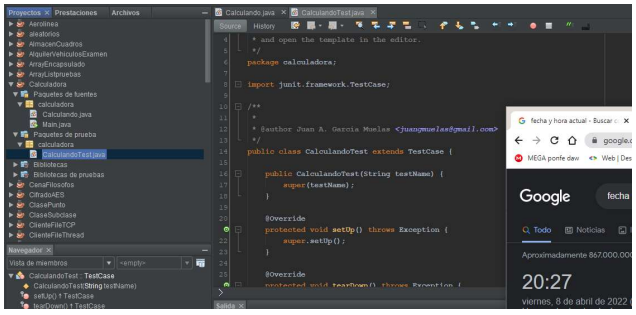
Detalle menú contextual Create/update test



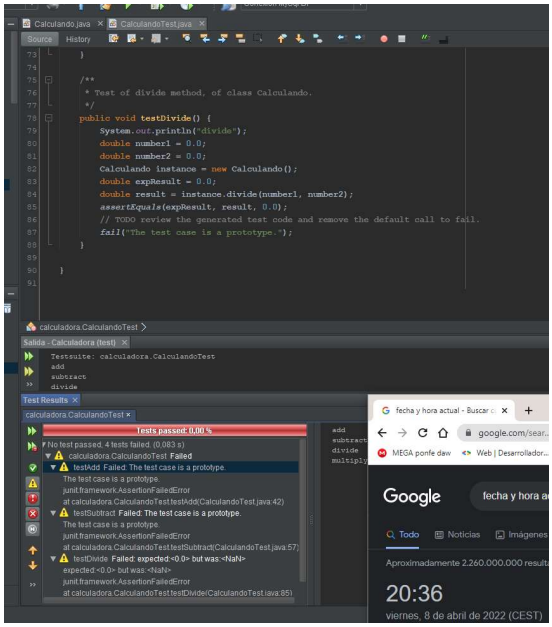
Selección de framework y código a generar.

2. Selecciona la nueva clase de pruebas que has generado. Ejecútala. Realiza una captura de la ventana Test results como solución a este aparado.

En nuestro paquete de pruebas se encuentra el archivo generado, y tras ejecutarlo, veremos que no pasan los test.



Detalle clase CalculandoTest, generada anteriormente.



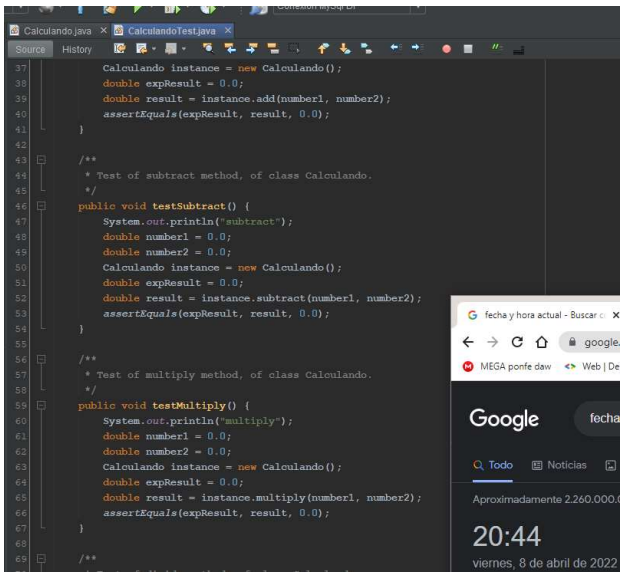
Detalle del error en la ejecución de los test.

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.

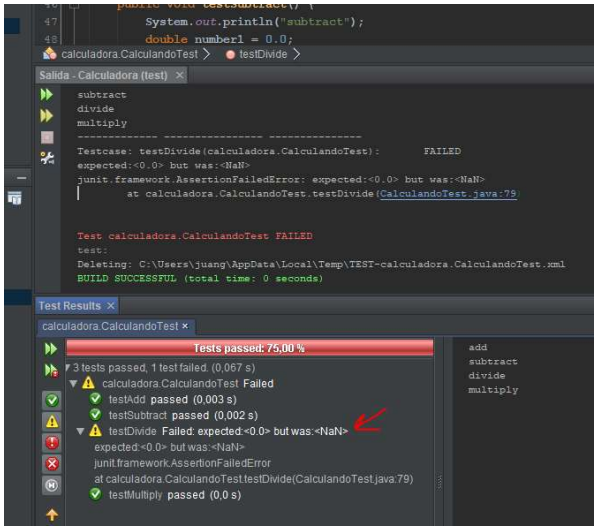
NetBeans ha generado estos métodos para los test. Tras los `assertEquals()` de cada método aparecía (como se aprecia en la captura del punto anterior) la línea con el `fail()`. Tras el borrado, quedan así.

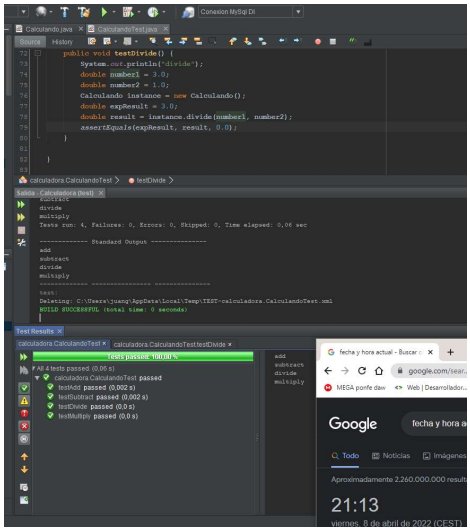


Detalle tras eliminación de fail();

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.

El test falla en el método `divide` al arrojar un `NaN`. Con aportarle un nuevo valor a las variables, corregimos el error.

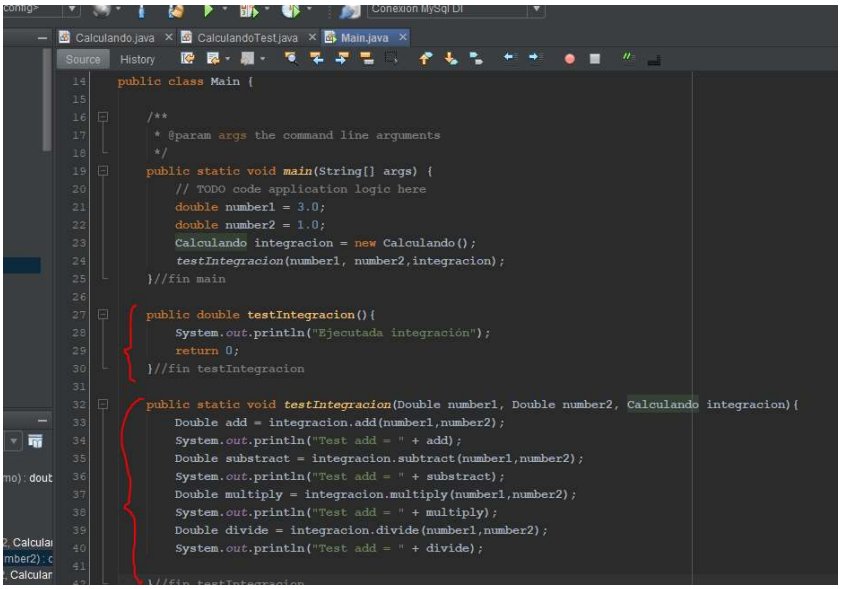




Detalle error test y solución.

5. Implementa la planificación de las pruebas de integración, sistema y regresión.

He elegido dos formas posibles para la prueba de integración.
La primera es creando un método en nuestra clase `main` que será llamado desde el test que genero a continuación en la clase `CalculandoTest`, para que Junit analice su ejecución.
La segunda opción verifica al ejecutar el proyecto, el correcto funcionamiento en su salida por consola.



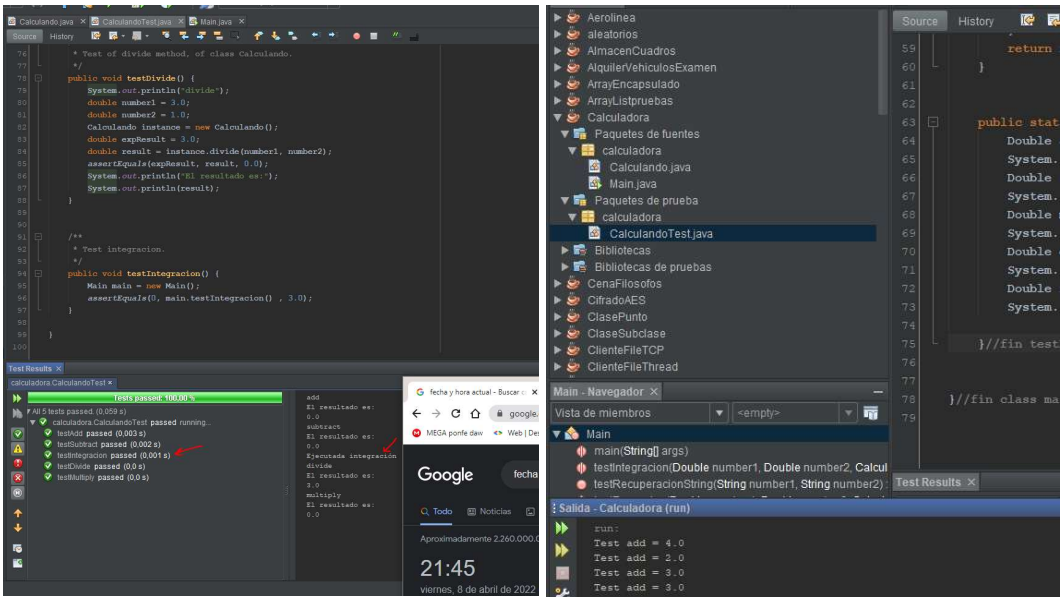
```java
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        double number1 = 3.0;
        double number2 = 1.0;
        Calculando integracion = new Calculando();
        testIntegracion(number1, number2,integracion);
    }//fin main

    public double testIntegracion(){
        System.out.println("Ejecutada integración");
        return 0;
    }//fin testIntegracion

    public static void testIntegracion(Double number1, Double number2, Calculando integracion){
        Double add = integracion.add(number1,number2);
        System.out.println("Test add = " + add);
        Double substract = integracion.subtract(number1,number2);
        System.out.println("Test add = " + substract);
        Double multiply = integracion.multiply(number1,number2);
        System.out.println("Test add = " + multiply);
        Double divide = integracion.divide(number1,number2);
        System.out.println("Test add = " + divide);
```

Opciones para prueba de integración en clase main.



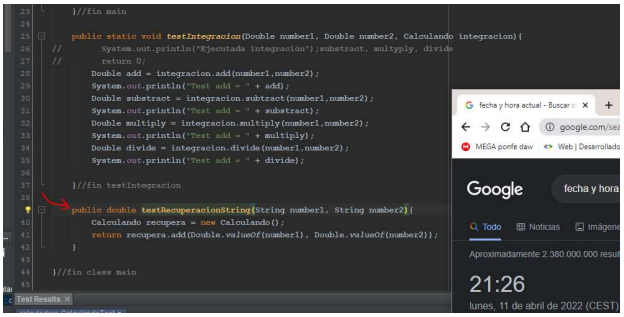
Detalle de Test Results positivo tras prueba de integración y de salida por consola.

Las pruebas de sistema, en buena parte se han visto al ejecutar las pruebas anteriores, pues los cálculos son correctos, la salida de datos esperada y con una velocidad menor de un segundo para las pruebas, por lo que valoro que se ejecuta de forma rápida.

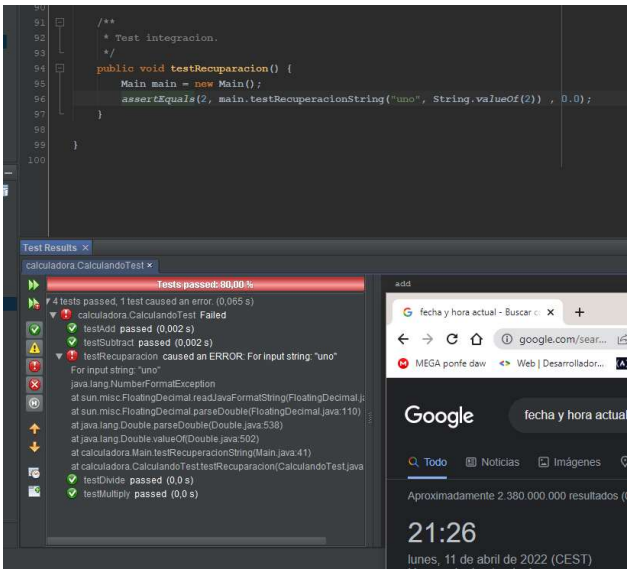
Realizo alguna prueba para ver cómo se recupera tras posibles entradas erróneas o no esperadas.

La prueba de recuperación la planteo sobre la posibilidad de que el usuario ingrese una cadena de texto en vez de un valor numérico.

Como puede observarse tras ejecutar los test, falla y nos muestra un error al intentar convertir el dato de texto.

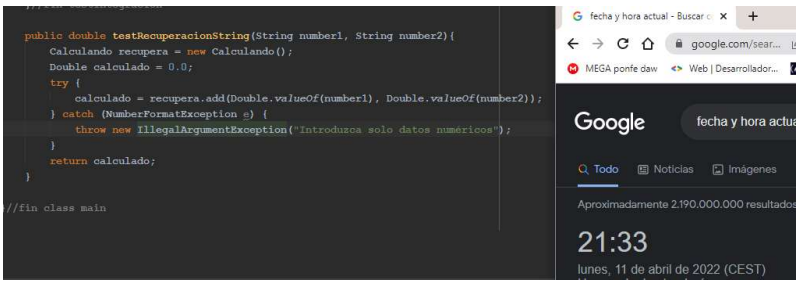


Método para testRecuperacionString().

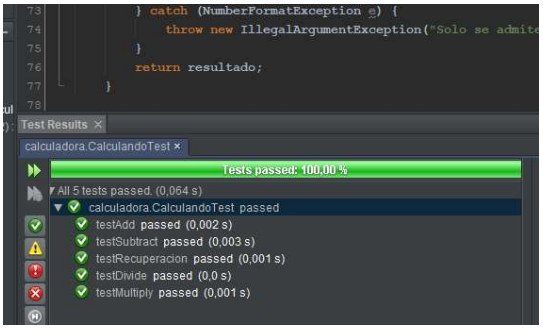


Detalle error en el Test Results.

Si el error está en cuando nos lanza una excepción `NumberFormatException`, lo solucionamos con la captura de esta en un bloque try-catch, que nos arrojará una excepción `IllegalArgumentException` con el mensaje que queramos mostrar sobre él.

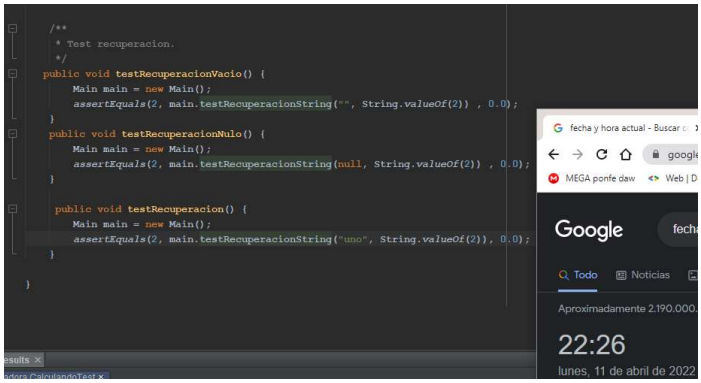


Modificación en nuestro test para recoger la excepción.



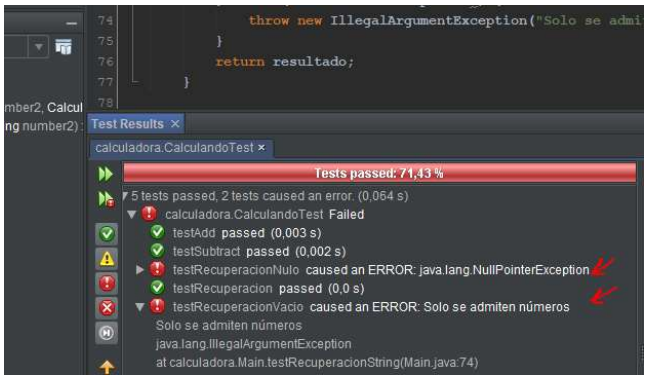
Test positivos tras recoger la excepción.

Forzamos nuevos posibles errores con campos vacíos o nulos, mediante nuevos métodos que traten de forma similar al anterior la entrada incorrecta de valores.



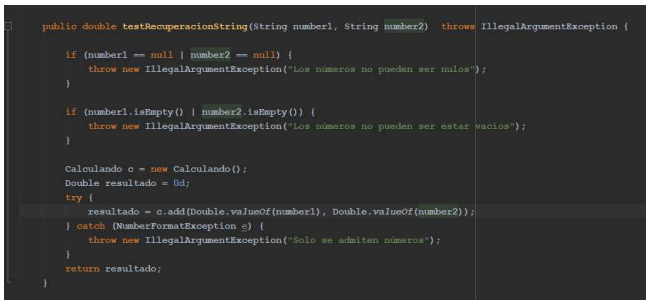
Detalle testRecuperacion para valores vacíos o nulos.

Como era de esperar, arrojará errores en la ejecución

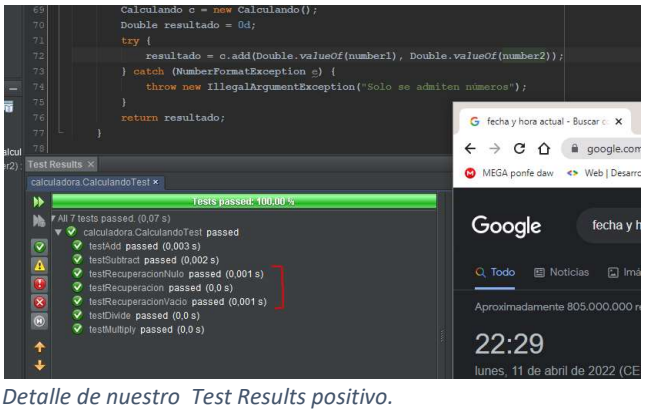


Detalle error tras ejecutar.

Se controlan las excepciones de forma similar a como lo hicimos con el primer caso de prueba.



Tratamiento de excepciones para recuperación.



Detalle de nuestro Test Results positivo.

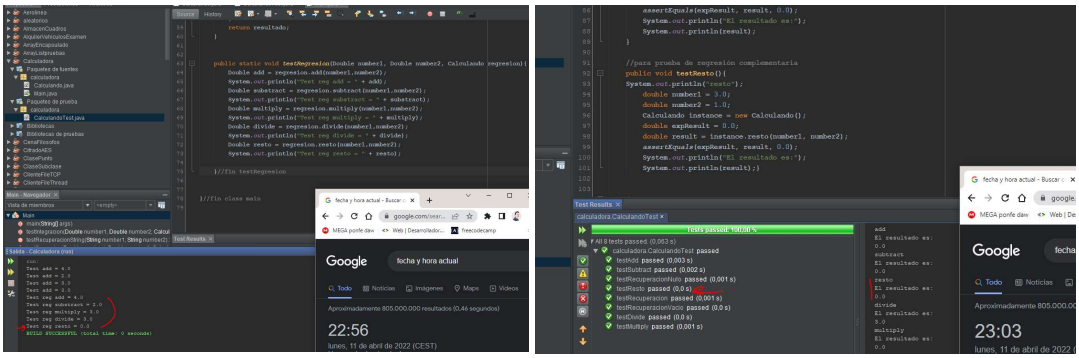
La prueba de regresión, también se ha pasado en varias ocasiones según se avanzaba con la tarea, al hacer distintas modificaciones para comprobar posibles errores.

Aún así y para reflejarlo de forma más directa en nuestros test, he creado un nuevo método que calcula el resto de la división entre los valores.

Tras ello, he realizado una nueva ejecución desde nuestra clase `main`, para ver su correcta salida.

```
27
28      public double divide(double number1, double number2){
29      return number1 / number2; }
30
31      //para prueba de regresión complementaria
32      public double resto(double number1, double number2){
33      return number1 % number2; }
34
35  }
36
```

Detalle método para resto.



Detalle salida correcta en consola y ejecución de test.

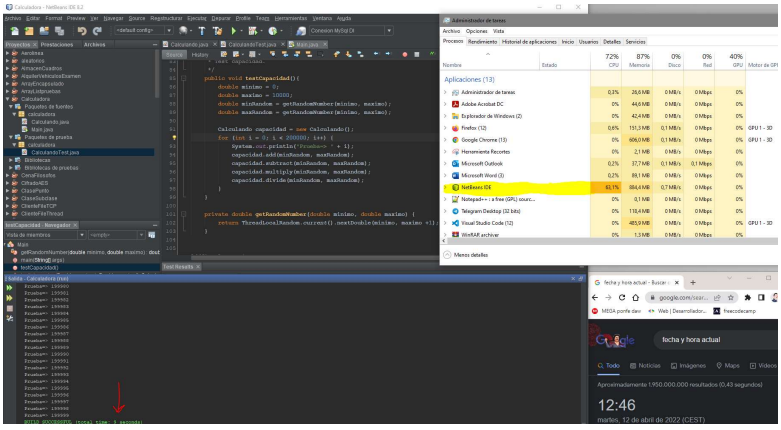
6. Planifica las restantes pruebas, estableciendo qué parámetros se van a analizar.

Nos quedan por realizar las siguientes pruebas:

Funcionales. Son pruebas que hemos superado en los test unitarios o al analizar valores extremos.

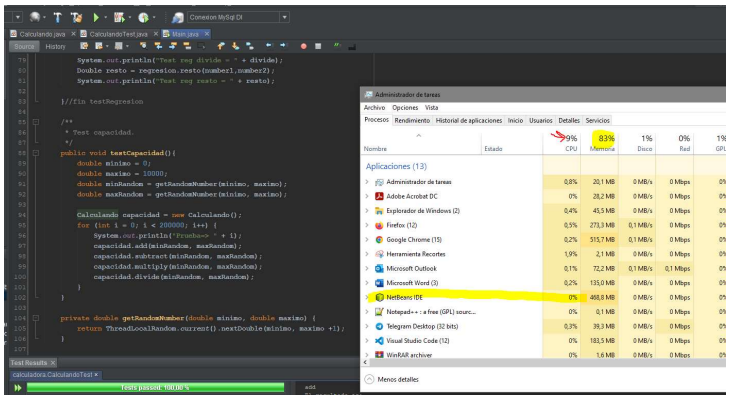
Capacidad y rendimiento. Mediante una nueva función llamada `testCapacidad()`, le pasaremos dos valores aleatorios entre el 0 y el 10.000 durante 200.000 ciclos a los métodos del test.

Aunque solo nos pide el enunciado la planificación de decidido ejecutarlo, y podemos observar como es capaz de hacer 200.000 operaciones matemáticas en 9 segundos (± 22.222/seg) sin bloqueo.



Prueba de capacidad y uso de recursos.

Recursos o eficiencia. Si comparamos la imagen inferior con la del punto anterior, se pude observar como hay un uso de memoria muy similar antes o durante la ejecución, y que solo se ve alterado el uso de CPU durante un corto espacio de tiempo y apenas superando la mitad de los recursos posibles, ante una petición desproporcionada.



Prueba de seguridad y uso de recursos.

Seguridad. Al ser una calculadora, como mucho podemos intentar hacerla más segura con una mayor restricción de permisos a los métodos y funciones (la mayoría `public`, en el código entregado).

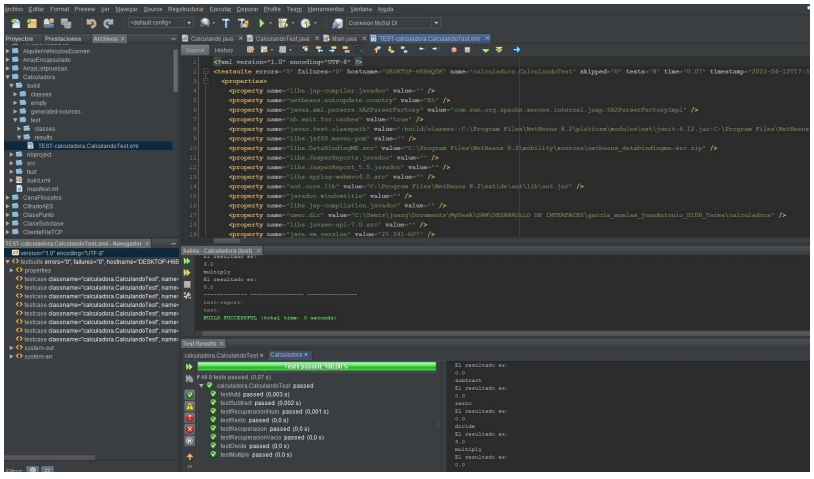
Usuario. Debe asegurar que la interfaz es amigable, intuitiva y que funciona correctamente.

Deberá el usuario por tanto, ser capaz de utilizarla con un número pequeño de acciones sencillas.

Aceptación. Son realizadas por el cliente final sobre el sistema completo. Poniéndome en esa situación, entiendo que en esta aplicación deben recoger y devolver de forma correcta los parámetros recogidos a través de las funciones (`add`, `substract`, `multiply`, `divide` y `resto`). Lo hemos comprobado mediante pruebas unitarias en los métodos, durante los puntos anteriores.

7. Supuestas exitosas las pruebas, documenta el resultado

Si bien he intentado documentar cada paso en los puntos anteriores, he considerado que era también positivo poder tener a nuestra disposición una prueba documental de nuestros test. Por ello, situados en nuestro proyecto `Calculadora`, a través de `Ejecutar/probar proyecto`, NetBeans ha realizado los test y guardado en el archivo `TEST-calculadora.CalculandoTest.xml` todos los datos recogidos durante la ejecución.



Detalle creación archivo Xml de nuestros test.

Como resumen complementario, podemos afirmar que:

En la prueba de capacidad y rendimiento, la aplicación es capaz de resolver de forma rápida y eficaz (9 seg) una elevada cantidad de operaciones matemáticas, con un uso más que razonable de memoria.

En la prueba de eficiencia y recursos, aplicada sobre el test anterior de sobrecarga y estrés, el cálculo de más de 22.222 operaciones por segundo y con un uso puntual de CPU, hacen valorar como positivo y correcto el funcionamiento de la aplicación.

Las pruebas funcionales han validado el comportamiento de la aplicación tras analizar el uso de valores límite y aleatorios.

Las pruebas unitarias y de regresión han funcionado de manera exitosa tras las modificaciones en peticiones y valores.

Las pruebas de sistema y de integración, mediante los test de recuperación han tenido un resultado correcto tras recoger las distintas excepciones tras una entrada de valores inesperada.