

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

Planifica la estrategia de pruebas, distinguiendo las tres partes que se han visto en la unidad. Para ello:

---

**1. Utiliza la herramienta de pruebas JUnit para realizar las pruebas sobre los métodos de la clase "Calculando" del proyecto "Calculadora". Guarda el resultado y comenta en qué ha consistido el trabajo y los resultados obtenidos. Hay que tener en cuenta que la prueba será exitosa siempre que le des valores numéricos a las variables.**

Haremos pruebas funcionales con Junit.

- **Análisis de valores límite:**

0, null, -10000000000, +10000000000, "abc", ""

(Nota, en nuestro caso no podemos usar null porque no se podría convertir a double).

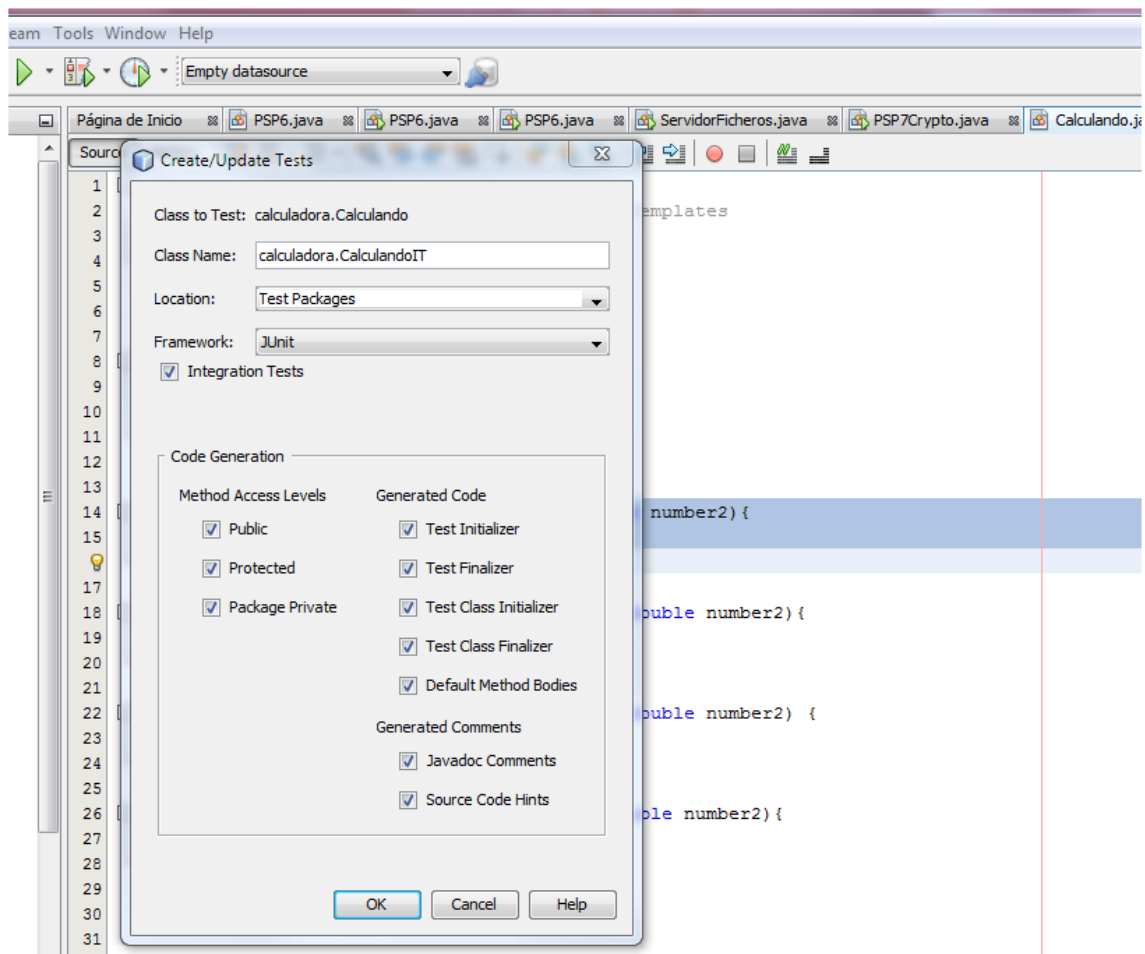
- **Particiones equivalentes:**

En particiones equivalentes usaremos como valor el 100 y el 1000

- **Pruebas aleatorias:**

Finalmente haremos un par de pruebas con valores totalmente random.

Para esto ejecutaremos la opción create/update tests dentro de tools, seleccionaremos las opciones deseadas (En nuestro caso tests de integración). Y pinchamos en ok.



Un test inicialmente es este código:

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

Para que sea funcional asignaremos valores a las variables (Los valores comentados en los puntos anteriores) y eliminaremos la llamada a fail que hay al final. También es posible reproducir esta clase con otro nombre para hacer tests acumulativos y poder lanzarlos todos juntos,

Debemos ajustar los test para que queden como esto:

```

//TEST SUMA HIGH NEGATIVES
@Test
public void testAddNegative() {
    System.out.println("add");
    double number1 = -1000000000.0;
    double number2 = -1000000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.add(number1, number2);
    assertEquals(expResult, result, -2000000000.0);
}

```

Dando valores a los variables y en assertEquals definiendo el resultado.

(NOTA: Es posible agregar más assertEquals para indicar margen de error admitido en la prueba, un string, u otras cuestiones).

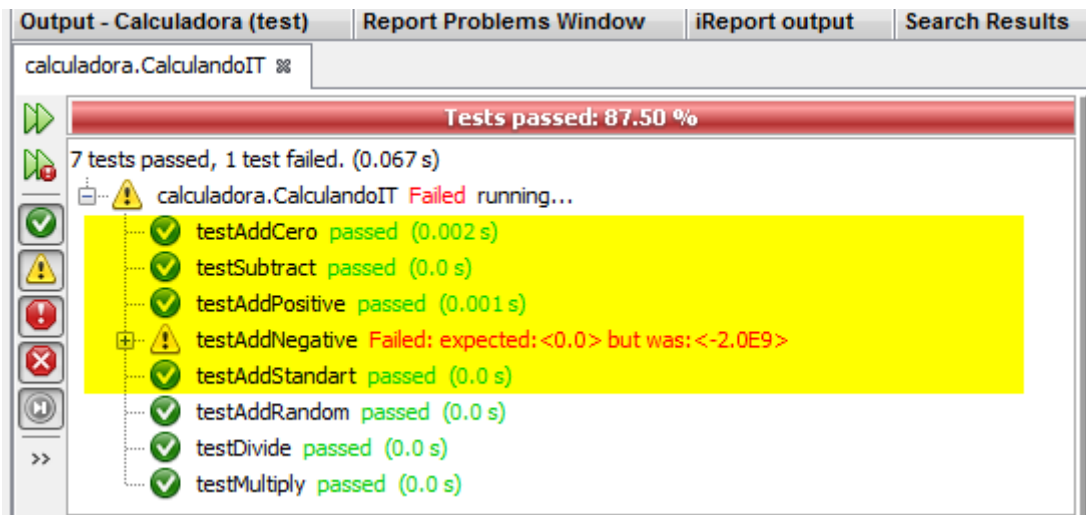
Ahora vamos a lanzar la “batería de tests” para el método add.

```

12 public void testAddCero() {
13     System.out.println("add");
14     double number1 = 0;
15     double number2 = 0;
16     Calculando instance = new Calculando();
17     double expectedResult = 0.0;
18     double result = instance.add(number1, number2);
19     assertEquals(expResult, result, 0.0);
20 }
21
22 //TEST SUMA HIGH NEGATIVES
23 @Test
24 public void testAddNegative() {
25     System.out.println("add");
26     double number1 = -1000000000.0;
27     double number2 = -1000000000.0;
28     Calculando instance = new Calculando();
29     double expectedResult = 0.0;
30     double result = instance.add(number1, number2);
31     assertEquals(expResult, result, -2000000000.0);
32 }
33
34 //TEST SUMA HIGH POSITIVES
35 @Test
36 public void testAddPositive() {
37     System.out.println("add");
38     double number1 = 1000000000.0;
39     double number2 = 1000000000.0;
40     Calculando instance = new Calculando();
41     double expectedResult = 0.0;
42     double result = instance.add(number1, number2);
43     assertEquals(expResult, result, 2000000000.0);
44 }
45
46 //TEST SUMA VALORES STANDART
47 @Test
48 public void testAddStandart() {
49     System.out.println("add");
50     double number1 = 100.0;
51     double number2 = 100.0;
52     Calculando instance = new Calculando();
53     double expectedResult = 0.0;
54     double result = instance.add(number1, number2);
55     assertEquals(expResult, result, 200.0);
56 }
57
58 //TEST SUMA VALORES RANDOM
59 @Test
60 public void testAddRandom() {
61     System.out.println("add");
62     double number1 = 8.0;
63     double number2 = 9800.0;
64     Calculando instance = new Calculando();
65     double expectedResult = 0.0;
66     double result = instance.add(number1, number2);
67     assertEquals(expResult, result, 9808.0);
68 }

```

Observamos que todos los test menos uno pasan correctamente. El que no pasa es bastante curioso y ahora veremos por qué.



Ejecutando una simple suma de negativos en una calculadora vemos que debería dar un numero negativo grande y lo que nos está volcando esta prueba es un error. Tocaría revisar porqué este método devuelve 0 cuando se suman dos negativos.

$$(-1) + (-1) = -2$$

QUE DEBEMOS HACER PARA QUE ESTE TEST PASE:

```

@Test
public void testAddNegative() {
    System.out.println("add");
    double number1 = -1000000000;
    double number2 = -1000000000;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.add(-number1, number2);
    assertEquals(expResult, result, -2000000000.0);
}
//TEST SUMA HIGH POSITIVES

```

Explicando la lógica:

El primer valor es negativo, por tanto instance.add (valornegativo)

El segundo valor se RESTARA un valor NEGATIVO (- x -) por tanto el segundo valor debe ser positivo (+number2)

Si lanzamos este test tal y como está arriba, PASARÁ.

```

testAddCero passed (0.001 s)
testAddPositive passed (0.0 s)
testAddNegative passed (0.0 s)
testAddStandart passed (0.0 s)

```

Repetimos el mismo proceso para el resto de los métodos, **ahora la resta**.

NOTA: He ido comentando el código para ir haciendo pruebas de forma independiente.

```

@Test
public void testSubtractZero() {
    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 RESTA HIGH NEGATIVE NEGATIVE
@Test
public void testSubtractNegative() {
    System.out.println("subtract");
    double number1 = -10000000000.0;
    double number2 = -10000000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.subtract(number1, number2);
    assertEquals(expectedResult, result, 0.0);
}

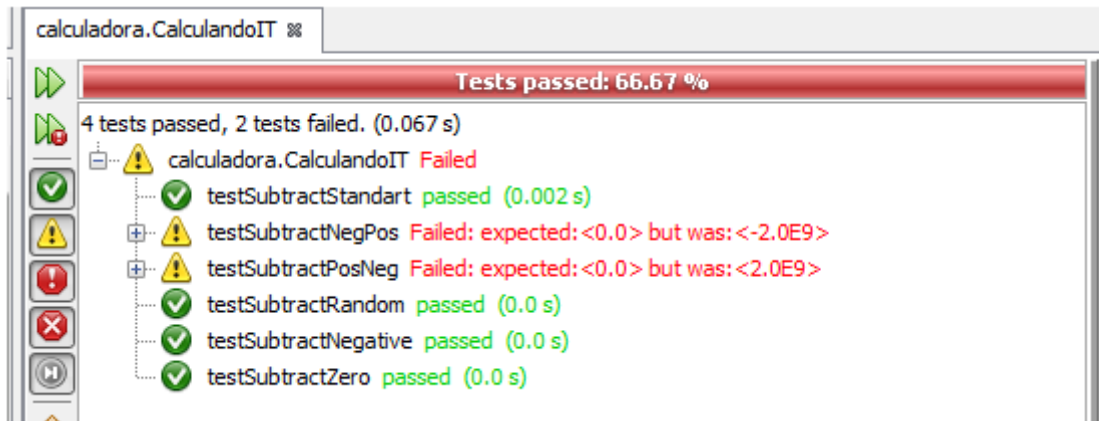
//TEST RESTA HIGH NEGATIVE POSITIVE
@Test
public void testSubtractNegPos() {
    System.out.println("subtract");
    double number1 = -10000000000.0;
    double number2 = 10000000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.subtract(number1, number2);
    assertEquals(expectedResult, result, -20000000000.0);
}

//TEST RESTA HIGH POSITIVE NEGATIVE
@Test
public void testSubtractPosNeg() {
    System.out.println("subtract");
    double number1 = 10000000000.0;
    double number2 = -10000000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.subtract(number1, number2);
    assertEquals(expectedResult, result, 0.0);
}

//TEST RESTA STANDART
@Test
public void testSubtractStandart() {
    System.out.println("subtract");
    double number1 = 100.0;
    double number2 = 50.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.subtract(number1, number2);
    assertEquals(expectedResult, result, 50.0);
}

//TEST RESTA RANDOM
@Test
public void testSubtractRandom() {
    System.out.println("subtract");
    double number1 = 218.0;
    double number2 = 68.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.subtract(number1, number2);
    assertEquals(expectedResult, result, 150.0);
}

```



Entiendo que estos resultados guardan relación con lo visto arriba y tienen la misma explicación.

Ahora analizaremos la multiplicación:

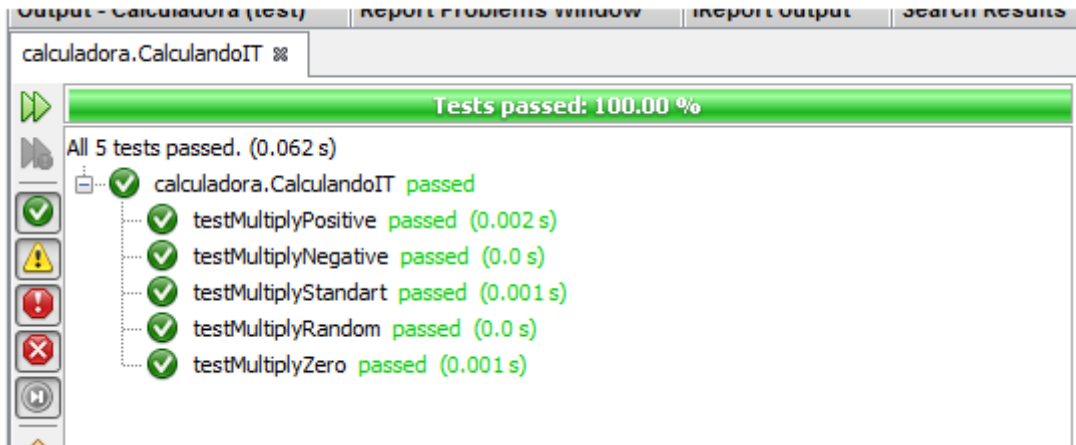
```
@Test
public void testMultiplyZero() {
    System.out.println("multiply");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result,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

@Test
public void testMultiplyPositive() {
    System.out.println("multiply");
    double number1 = 1000000.0;
    double number2 = 1000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result, 1000000000000.0);
}

@Test
public void testMultiplyNegative() {
    System.out.println("multiply");
    double number1 = -1000000.0;
    double number2 = -1000000.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result, 1000000000000.0);
}

@Test
public void testMultiplyStandart() {
    System.out.println("multiply");
    double number1 = 5.0;
    double number2 = 5.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result, 25.0);
}

@Test
public void testMultiplyRandom() {
    System.out.println("multiply");
    double number1 = 132.0;
    double number2 = 348.0;
    Calculando instance = new Calculando();
    double expectedResult = 0.0;
    double result = instance.multiply(number1, number2);
    assertEquals(expectedResult, result, 45936.0);
}
```



Finalmente la división.

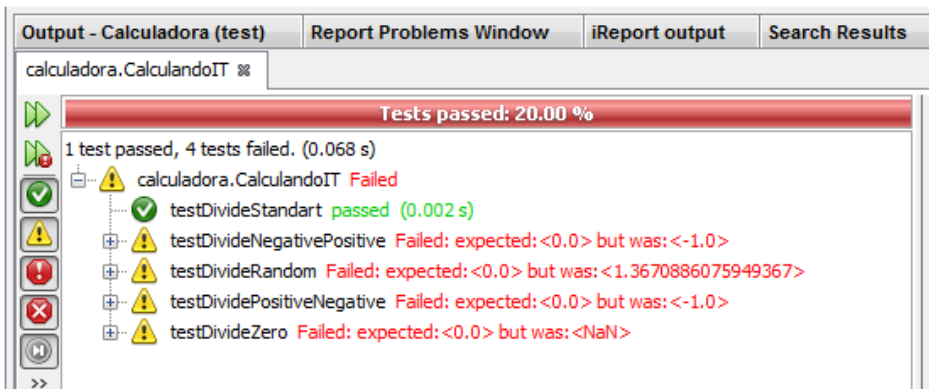
```
@Test
public void testDivideZero() {
    System.out.println("divide");
    double number1 = 0.0;
    double number2 = 0.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, 0);
}

@Test
public void testDividePositiveNegative() {
    System.out.println("divide");
    double number1 = 1000000000.0;
    double number2 = -1000000000.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, -1.0);
}

@Test
public void testDivideNegativePositive() {
    System.out.println("divide");
    double number1 = -1000000000.0;
    double number2 = 1000000000.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, -1.0);
}

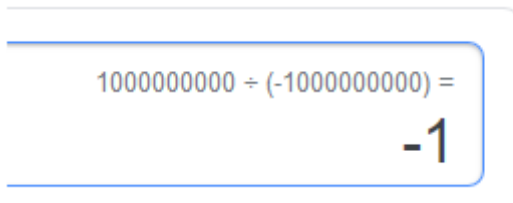
@Test
public void testDivideStandart() {
    System.out.println("divide");
    double number1 = 100.0;
    double number2 = 4.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, 25.0);
}

@Test
public void testDivideRandom() {
    System.out.println("divide");
    double number1 = 324.0;
    double number2 = 237.0;
    Calculando instance = new Calculando();
    double expResult = 0.0;
    double result = instance.divide(number1, number2);
    assertEquals(expResult, result, 1.3670886075949367);
}
```



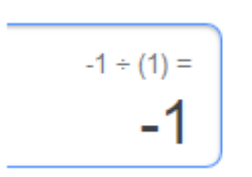
Consideraciones: Las mismas que arriba:

1. Los métodos “positivo entre negativo dan un resultado incorrecto. Aquí un ejemplo.



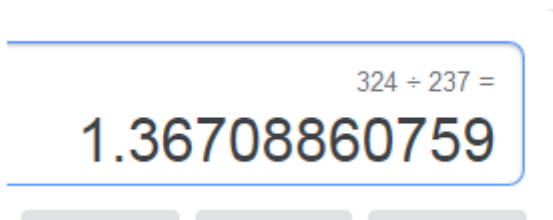
1000000000 ÷ (-1000000000) =  
-1

2. En el caso de negativo entre positivo, ocurre el mismo error.



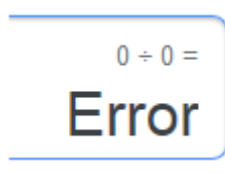
-1 ÷ (1) =  
-1

3. El caso de la división random, el error ocurre por el redondeo decimal (Espera un mayor número de decimales y al no encontrarlo da un error).



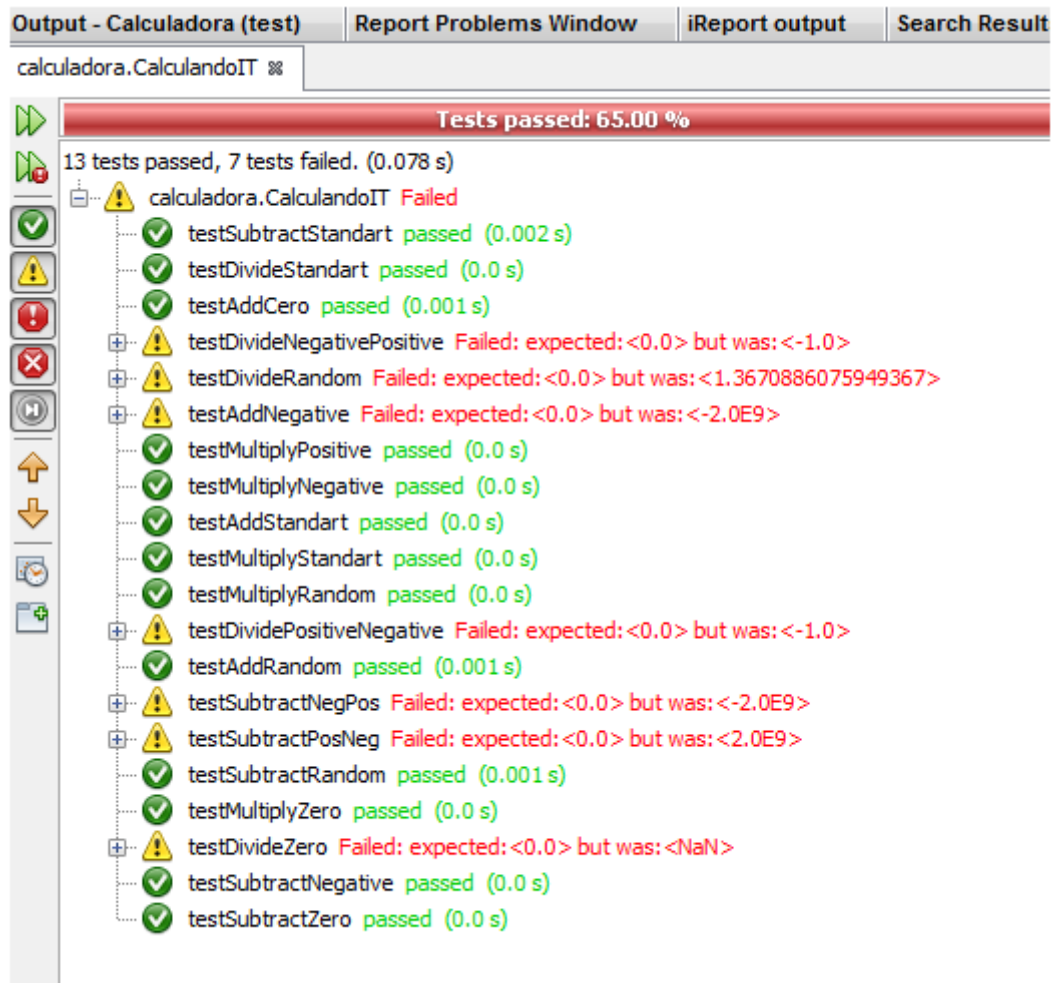
324 ÷ 237 =  
1.36708860759

4. La división entre ceros está correcta (Cualquier numero dividido entre 0 da error).



0 ÷ 0 =  
Error





**Conclusiones de estas pruebas. Se deben revisar los métodos add, subtract y divide, parece ser que no computan bien la simbología + y – y generan errores cuando se trabaja e intercalan números negativos.**

Nota: Hay posibilidades que no se han llegado a cubrir en las pruebas (Intercalado de negativos y positivos en todos los métodos).

## **2. Implementa la planificación de las pruebas de integración, sistema y regresión.**

### **Integración ascendente:**

Se empieza por los módulos de más bajo nivel hasta llegar al programa principal. Probando la relación entre los diferentes módulos (En esta tarea solo tenemos un módulo), ascendiendo e integrando a las pruebas cada vez más módulos hasta llegar al último nivel, que sería la aplicación completa con todos sus módulos. La integración es de abajo a arriba.

### **Integración descendente:**

Consiste en integrar los módulos moviéndose de arriba abajo por la jerarquía. Se lanzan pruebas por cada módulo agregado y se usa el módulo principal como controlador.

**Sistema configuración:**

El objetivo es verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas y determinar la configuración óptima del sistema.

Ejemplos: Tests de red si se va a ejecutar la aplicación vía remota, Revisar la correcta configuración de java, etc.

**Sistema recuperación:**

Se fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo de forma satisfactoria.

Habría que revisar:

- La inicialización.
- Los mecanismos de recuperación del estado del sistema.
- Los datos.
- El proceso de arranque.

**Regresión:**

Como resultado de esa modificación (o adición) de módulos, podemos introducir errores en el programa que antes no teníamos. Es necesario volver a probar la aplicación tras cada añadido. La prueba de regresión es 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.

En este caso solo tenemos un módulo. Si agregásemos otro tendríamos que repetir las pruebas implicándolo en el proceso.

**3. Planifica las restantes pruebas, estableciendo qué parámetros se van a analizar.****De capacidad.**

(Las clásicas pruebas de stress.) Se pueden realizar vía automatismos lanzando un montón de instancias de un programa de manera conjunta para “intentar estresar la aplicación” y ver su comportamiento. Lo mismo aplica para el sistema, la red, etc.

**De rendimiento.**

Aquí se debe monitorizar y comprobar el rendimiento utilizando herramientas de monitorización de hardware y software.

También se puede agregar código para revisar nuestros tiempos de ejecución y ver si son excesivos.

**De disponibilidad de recursos.**

Es una prueba de eficiencia, actualmente tiene menos importancia por la enorme cantidad de recursos hardware disponibles en las máquinas (Y va en aumento). Está

dejando paso a las pruebas de usabilidad que van ganando importancia con el paso del tiempo.

De seguridad.

Verifica que los mecanismos de protección incorporados en el sistema lo protegerán de accesos no autorizados.

Para ello:

- Revisar políticas de java.
- 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.
- Intentar proteger la aplicación de usos indebidos o accesos no autorizados.

**4. Supuestas exitosas las pruebas, documenta el resultado**

El resultado de todas estas pruebas y planificaciones es este documento y el proyecto con las pruebas que se adjuntará en la tarea.