

UT05. DISEÑO Y REALIZACIÓN DE PRUEBAS

Entornos de Desarrollo

1 DAW – C.I.F.P. Carlos III - Cartagena

5. Validaciones

Validaciones

- Descubrir errores, desde el punto de vista de los requisitos.
- Pruebas de caja negra que demuestran la conformidad con los requisitos

6. Pruebas de código

Pruebas de código

- **Objetivo:** encontrar errores
- Definir una serie de casos de prueba
- Enfoques ya estudiados
 - Prueba funcional de caja negra
 - Prueba estructural de caja blanca
 - Pruebas aleatorias

7. Normas de calidad

Normas de calidad

- Estándares **BSI**
 - BS 7925-1, Pruebas de software. Parte 1. Vocabulario.
 - BS 7925-2, Pruebas de software. Parte 2. Pruebas de los componentes software.
- Estándares IEEE de pruebas de software.:
 - IEEE estándar 829, Documentación de la prueba de software.
 - IEEE estándar 1008, Pruebas de unidad
 - Otros estándares **ISO** / IEC 12207, 15289
- Otros estándares sectoriales
 - ISO/IEC 29119 con todas las fases de las pruebas ➔
<http://softwaretestingstandard.org/>

8. Pruebas unitarias

JUnit

- Framework de pruebas unitarias creado por Erich Gamma y Kent Beck.
- Es una herramienta de código abierto.
- Posibilidad de crear informes en HTML.
- Organización de las pruebas en Suites de pruebas.
- Es la herramienta de pruebas más extendida para el lenguaje Java.
- Los entornos de desarrollo para Java, NetBeans y Eclipse, incorporan un plugin para JUnit.
- Herramienta para realizar pruebas unitarias automatizadas.
- Se realizan sobre una clase para probar su comportamiento de modo aislado independientemente del resto de clases de la aplicación

9. Automatización de la prueba

Automatización de la prueba

- Los entorno de desarrollo, integran frameworks, que permiten automatizar las pruebas.
- Una vez diseñados los casos de prueba, pasamos a probar la aplicación
- **JUnit**
 - Creación de la aplicación
 - En el .java o .class > botón derecho > Tools> Create/Update Test.
 - Se crea una clase de prueba en Test Packages cuyo código se ha de modificar.

Ejemplo

- Clase Calculadora

```
public class Calculadora {
```

```
    private int num1;  
    private int num2;
```

```
    public Calculadora(int a, int b) {  
        num1=a;  
        num2=b;  
    }
```

```
    public int suma() {  
        int result=num1+num2;  
        return result;  
    }
```

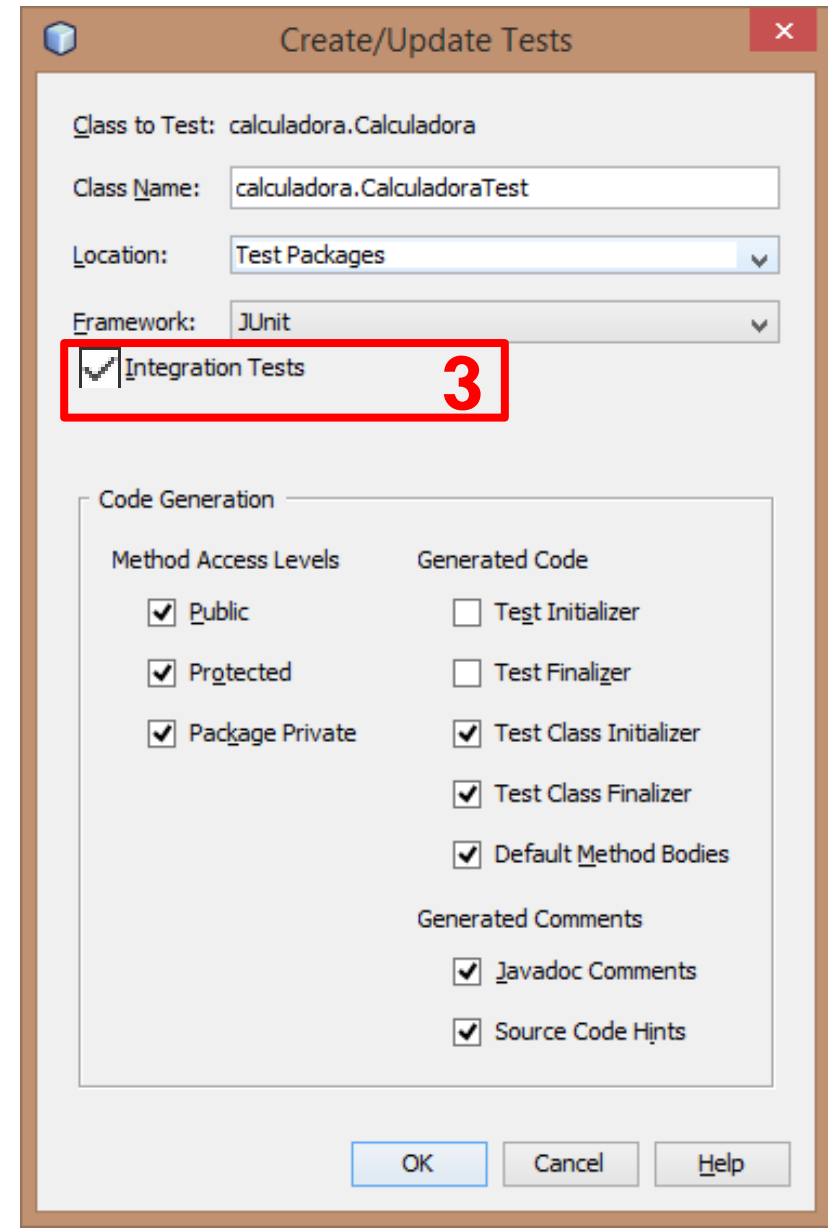
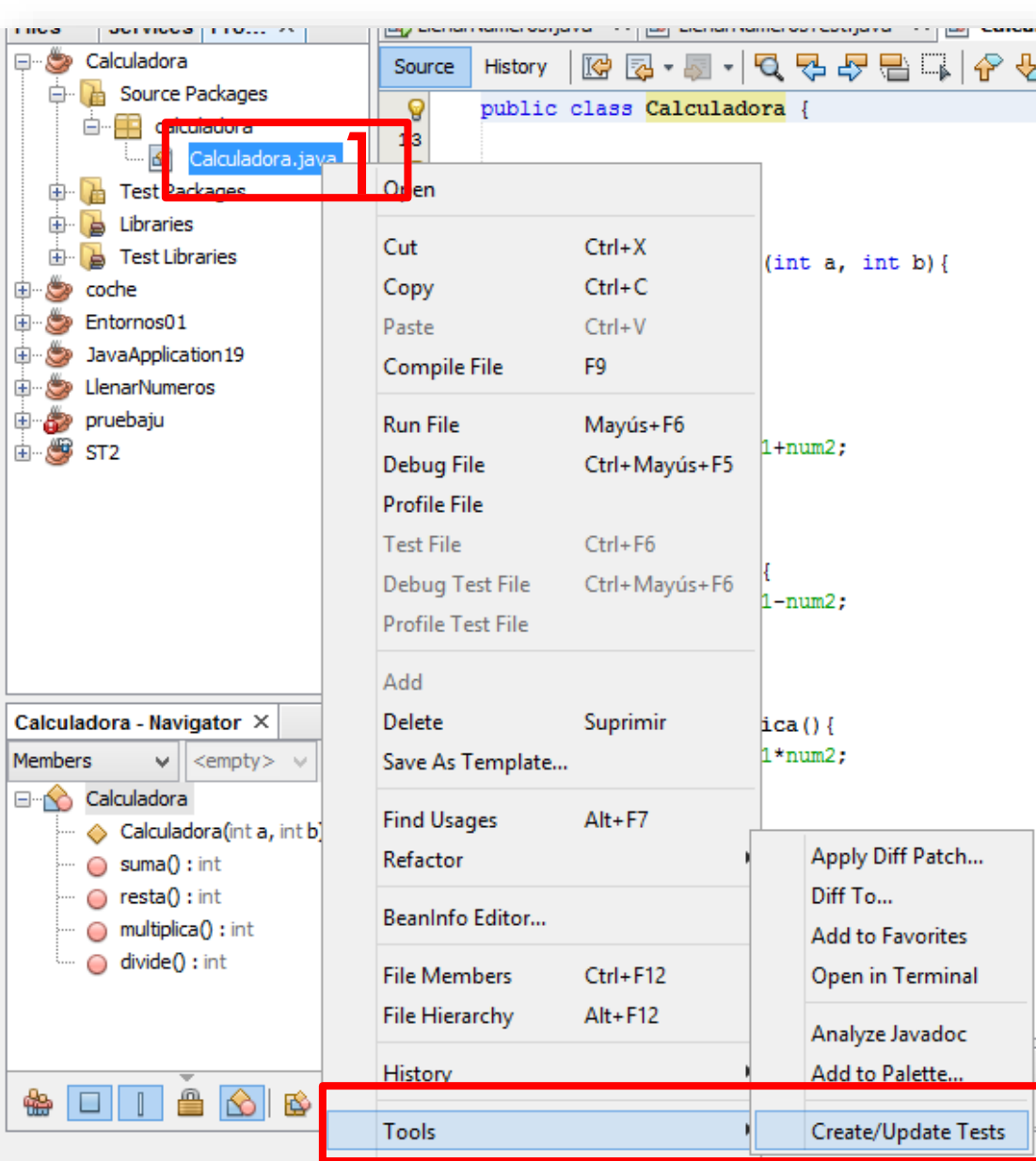
```
    public int resta() {  
        int result=num1-num2;  
        return result;  
    }
```

```
    public int multiplica() {  
        int result=num1*num2;  
        return result;  
    }
```

```
    public int divide() {  
        int result=num1/num2;  
        return result;  
    }
```

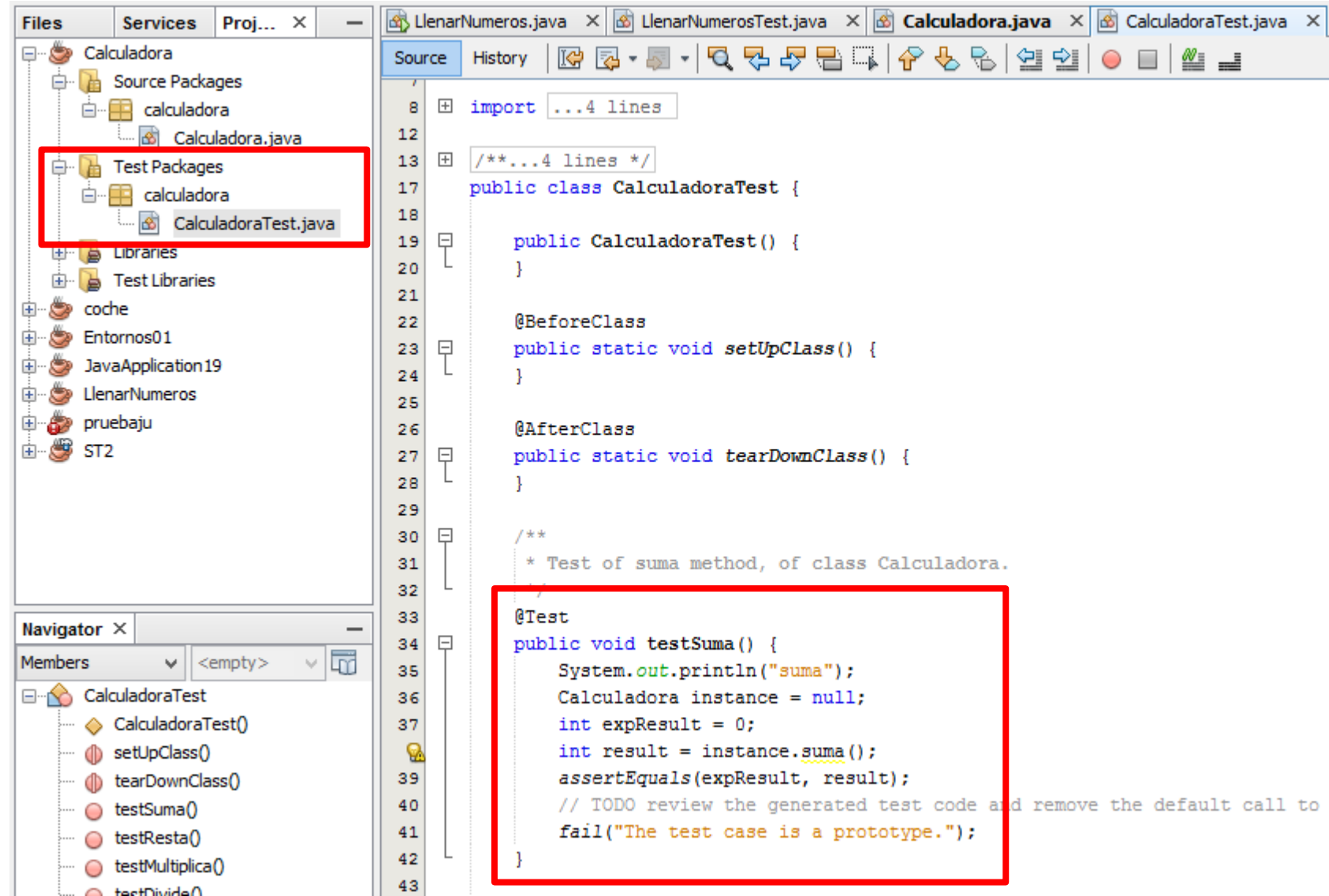
```
}
```

Creación de una clase prueba para Calculadora



Creación de una clase prueba

- Se crean 4 métodos de prueba
- Métodos **públicos**
- Ni devuelven, ni reciben información
- Nombre del método anteponiendo *test*
- **@Test** → indica al compilador que es un método prueba
- Llamada al método **fail()** con un mensaje que indica que es un prototipo, todavía no implementado



The screenshot displays an IDE with the following components:

- Files Explorer:** Shows a project structure with 'Test Packages' highlighted in red. Inside 'Test Packages', there is a 'calculadora' package containing 'CalculadoraTest.java'.
- Navigator:** Shows the 'CalculadoraTest' class with its methods: 'CalculadoraTest()', 'setUpClass()', 'tearDownClass()', 'testSuma()', 'testResta()', 'testMultiplica()', and 'testDivide()'.
- Source Editor:** Displays the source code of 'CalculadoraTest.java'. The code includes imports, package declarations, and class annotations. A red box highlights the 'testSuma()' method, which is a prototype test method.

```
8  import ...4 lines
12
13  /**...4 lines */
17  public class CalculadoraTest {
18
19      public CalculadoraTest() {
20      }
21
22      @BeforeClass
23      public static void setUpClass() {
24      }
25
26      @AfterClass
27      public static void tearDownClass() {
28      }
29
30      /**
31       * Test of suma method, of class Calculadora.
32       */
33
34      @Test
35      public void testSuma() {
36          System.out.println("suma");
37          Calculadora instance = null;
38          int expResult = 0;
39          int result = instance.suma();
40          assertEquals(expResult, result);
41          // TODO review the generated test code and remove the default call to
42          fail("The test case is a prototype.");
43      }
```

JUnit: etiquetas

- **@Test**: define un método test
- **@Before**: ejecuta el método antes de cada test
- **@After**: ejecuta el método después de cada test
- **@BeforeEach**: se ejecuta antes de cualquier método de prueba. Se puede utilizar para inicializar datos. Puede haber varios métodos con esta anotación. JUnit
- **@AfterEach**: se ejecuta después de la ejecución de cada método de prueba. Se puede utilizar para limpiar datos. Puede haber varios métodos en la clase de prueba con esta anotación. JUnit
- **@BeforeClass**: se ejecuta una vez antes de todos los tests
- **@AfterClass**: se ejecuta una vez después de todos los tests
- **@Test (expected = Exception.class)**: Falla si el método no lanza una excepción
- **@Ignore**: ignora el test

MÉTODOS	MISIÓN
assertTrue(boolean expresión) assertTrue(String mensaje, boolean expresión)	Comprueba que la expresión se evalúe a true. Si no es así y se incluye en el String , al producirse el error se lanzará el mensaje
assertFalse(boolean expresión) assertFalse(String mensaje, boolean expresión)	Comprueba que la expresión se evalúe a false. Si no es así y se incluye en el String , al producirse el error se lanzará el mensaje
assertEquals(valorEsperado, valorReal) assertEquals(String mensaje, valorEsperado, valorReal)	Comprueba que el valorEsperado es igual al valorReal . Si no son iguales y se incluye el String, se lanzará el mensaje. Permite comparar diferentes tipos.
assertNull(Object objeto) assertNull(String mensaje, Object objeto)	Comprueba que el objeto sea null . Si no es null y se incluye el String al producirse el error se lanzará el String
assertNotNull(Object objeto) assertNotNull(String mensaje, Object objeto)	Comprueba que el objeto no sea null . Si es null y se incluye el String al producirse el error se lanzará el String
assertSame(Object objetoEsperado, Object objetoReal) assertSame(String mensaje, Object objetoEsperado, Object objetoReal)	Comprueba que el objetoEsperado y objetoReal sean el mismo objeto. Si no son el mismo y se incluye el String, se lanzará el mensaje
assertNotSame(Object objetoEsperado, Object objetoReal) assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)	Comprueba que el objetoEsperado y objetoReal no sean el mismo objeto. Si son el mismo y se incluye el String , se lanzará el mensaje
fail() fail(String mensaje)	Hace que la prueba falle. Si se incluye un String , la prueba fallará lanzando el mensaje

Ejecución de la prueba

@Test

```
public void testSuma() {  
    Calculadora calculo=new Calculadora(20,10);  
    int expResult = 30;  
    int result = calculo.suma();  
    assertEquals(expResult, result);  
}
```

Test Libraries

- coche
- Entornos01
- JavaApplication19
- LlenarNumeros
- pruebaju
- ST2

```
30 /**  
31  * Test of suma method, of class Calculadora.  
32  */  
33 @Test  
34 public void testSuma() {  
35     Calculadora calculo=new Calculadora(20,10);  
36     int expResult = 30;  
37     int result = calculo.suma();  
38     assertEquals(expResult, result);  
39 }  
40  
41 /**  
42  * Test of resta method, of class Calculadora.  
43  */
```

calculadora.CalculadoraTest X

Tests passed: 25,00 %

1 test passed, 3 tests caused an error. (0,368 s)

- calculadora.CalculadoraTest Failed
 - testResta caused an ERROR: java.lang.NullPointerException
 - testMultiplica caused an ERROR: java.lang.NullPointerException
 - testDivide caused an ERROR: java.lang.NullPointerException

resta
multiplica
divide

testResta - Navigator X

Members

CalculadoraTest

- CalculadoraTest()
- setUpClass()
- tearDownClass()
- testSuma()
- testResta()
- testMultiplica()
- testDivide()

Watches

Test Results X

Output

calculadora.CalculadoraTest X

Tests passed: 25,00 %

1 test passed, 3 tests caused an error. (0,368 s)

calculadora.CalculadoraTest Failed

resta
multiplica
divide

Fallo

Ejecución de la prueba

- Modifica el resto de los métodos para realizar la prueba

```
@Test
public void testResta() {
    Calculadora calculo=new Calculadora(20,10);
    int expectedResult = 10;
    int result = calculo.resta();
    assertEquals(expectedResult, result);
}

/**
 * Test of multiplica method, of class Calculadora.
 */
@Test
public void testMultiplica() {
    Calculadora calculo=new Calculadora(20,10);
    int expectedResult = 200;
    int result = calculo.multiplica();
    assertEquals(expectedResult, result);
}

/**
 * Test of divide method, of class Calculadora.
 */
@Test
public void testDivide() {
    Calculadora calculo=new Calculadora(20,10);
    int expectedResult = 2;
    int result = calculo.divide();
    assertEquals(expectedResult, result);
}
```

```
44 @Test
45 public void testResta() {
46     Calculadora calculo=new Calculadora(20,10);
47     int expectedResult = 10;
48     int result = calculo.resta();
49     assertEquals(expectedResult, result);
50 }
51
52 /**
53  * Test of multiplica method, of class Calculadora.
54  */
55 @Test
56 public void testMultiplica() {
57     Calculadora calculo=new Calculadora(20,10);
58     int expectedResult = 200;
59     int result = calculo.multiplica();
60     assertEquals(expectedResult, result);
61 }
```

Watches Test Results × Output

calculadora.CalculadoraTest ×

Tests passed: 100,00 %

All 4 tests passed. (0,117 s)

Fallo vs error

Fallo → modifica el método para que, en *multiplica()*, el resultado esperado no coincida con el real y añade un mensaje de aviso.

Error → asignamos 0 al segundo parámetro para el método *divide()*

calculadora.CalculadoraTest ×

Tests passed: 50,00 %

2 tests passed, 1 test failed, 1 test caused an error. (0,106 s)

- calculadora.CalculadoraTest Failed
 - testResta passed (0,003 s)
 - testSuma passed (0,0 s)
 - testMultiplica Failed: Fallo en multiplica expected: <20> but was: <200>
 - Fallo en multiplica expected: <20> but was: <200>
 - junit.framework.AssertionFailedError
 - at calculadora.CalculadoraTest.testMultiplica(CalculadoraTest.java:60)
 - testDivide caused an ERROR: / by zero
 - / by zero
 - java.lang.ArithmeticException
 - at calculadora.Calculadora.divide(Calculadora.java:38)
 - at calculadora.CalculadoraTest.testDivide(CalculadoraTest.java:70)

Ejecución de la prueba

- Modifica el **divide()** para que pase la prueba.
- Modifica el método **resta()** y añade **resta2()** y **divide2()**. Crea después los test para probar los tres métodos.
- Utiliza los métodos **assertTrue**, **assertFalse**, **assertNull**, **assertNotNull** o **assertEquals**, según convenga.

```
public int resta(){
    int resul;
    if(resta2()) resul = num1 - num2;
    else        resul = num2 - num1;
    return resul;
}

public boolean resta2(){
    if (num1 >= num2) return true;
    else              return false;
}

public Integer divide2(){
    if(num2==0) return null;
    int resul = num1 / num2;
    return resul;
}
```

Lanzar un método con excepciones

- Para probar un método que puede lanzar excepciones se utiliza el parámetro **expected** con la anotación **@Test**.
- Ejemplo en **divide0** → la prueba fallará si no se produce la excepción

```
public int divide0(){
    if(num2==0)
        throw new java.lang.ArithmeticException("División por 0");
    else{
        int resul=num1/num2;
        return resul;
    }
}
```

```
/**
 * Test of divide0 method, of class Calculadora2.
 */
@Test(expected = java.lang.ArithmeticException.class)
public void testDivide0() {
    Calculadora2 calculo = new Calculadora2(20,0);
    Integer resultado = calculo.divide0();
}
}
```

Tipos de anotaciones

- **@Before**

- Código que será ejecutado antes de cualquier método de prueba.
- Se puede utilizar para inicializar datos.
- Puede haber varios métodos en la clase de prueba con esta anotación.

- **@After**

- El código será ejecutado después de la ejecución de todos los métodos de prueba.
- Se puede utilizar para limpiar datos. Puede haber
- Puede haber varios métodos en la clase de prueba con esta anotación.

Ejemplo de Calculadora con @Before y @After

```
public class Calculadora2Test {  
    private Calculadora2 calcu;  
    private int resultado;  
  
    public Calculadora2Test() {  
    }  
  
    @Before  
    public void creaCalculadora2() {  
        calcu=new Calculadora2 (20, 10);  
    }  
  
    @After  
    public void borraCalculadora2() {  
        calcu=null;  
    }  
}
```

Antes de ejecutar cualquier método de prueba (suma, resta...), ejecuta @Before .
Al finalizar los test, ejecuta los métodos @After

@BeforeClass @AfterClass

- **@BeforeClass /@AfterClass**

- Solo puede haber un método con esas etiquetas.
- Es invocado una vez al principio/final del lanzamiento de todas las pruebas.
- Se suele utilizar para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerable en ejecutarse.

Pruebas parametrizadas

- Ejecutar una prueba varias veces con distintos valores de entrada.
 - Añadir
 - la etiqueta `@RunWith(Parameterized.class)` a la clase prueba,
 - un atributo por cada uno de los parámetros de la prueba y
 - un constructor con argumentos = atributos a probar.
 - Suma → 3 parámetros (dos datos de entrada y un resultado)
 - Definir la etiqueta `@Parameters`, encargado de devolver la lista de valores a probar.
 - Filas de valores para num1, num2 y resul → {20,10,30}


```
import java.util.Arrays;
import java.util.Collection;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
```

```
/**...4 lines */
```

```
@RunWith(Parameterized.class)
```

```
public class Calculadora4Test {
```

```
    private int num1;
    private int num2;
    private int resul;
```

```
    public Calculadora4Test(int num1, int num2, int resul) {
        this.num1=num1;
        this.num2=num2;
        this.resul=resul;
    }
```

```
@Parameters
```

```
public static Collection<Object[]> numeros() {
    return Arrays.asList(new Object [][]{{20,10,30},{3,2,5},{2,2,2}});
}
```

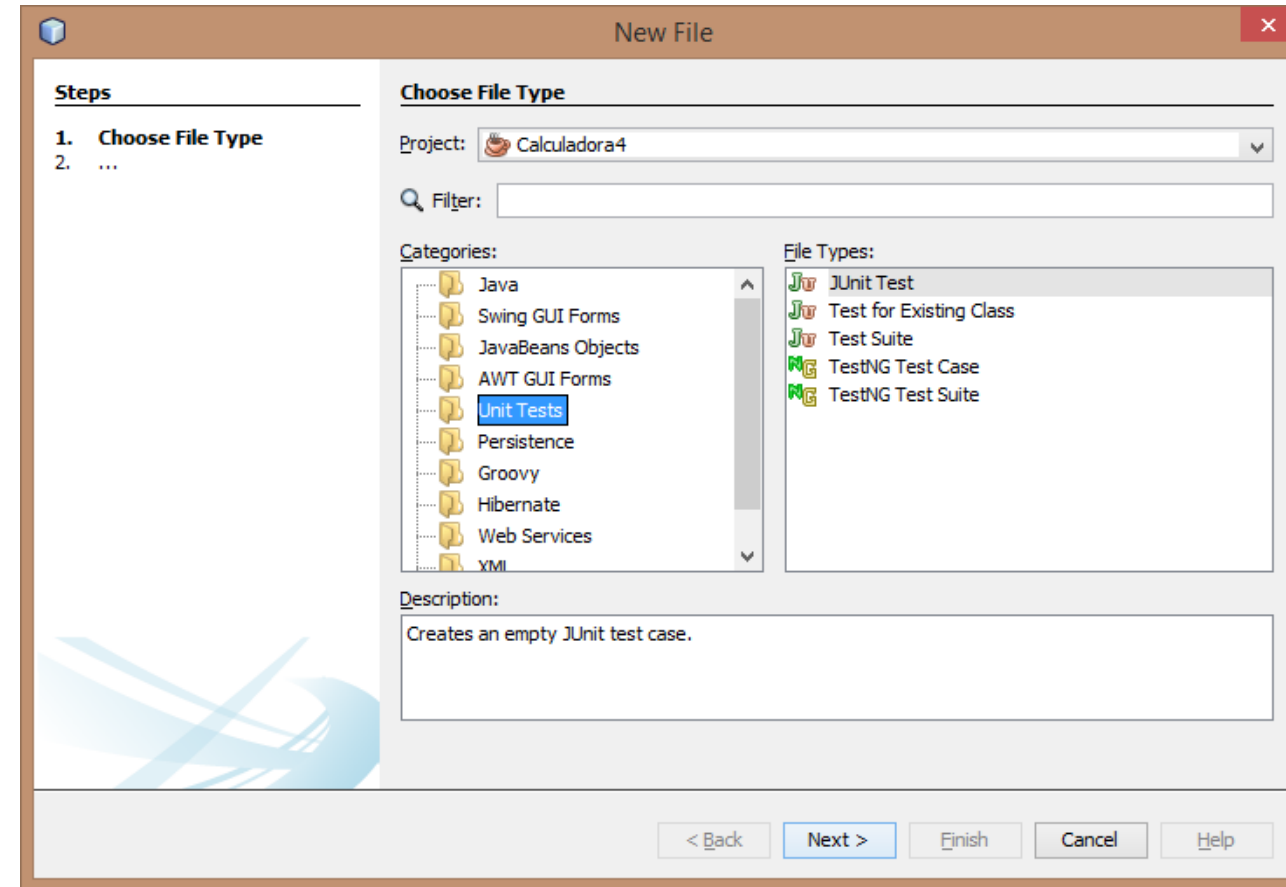
```
/**
 * Test of suma method, of class Calculadora4.
 */
@Test
public void testSuma() {
    System.out.println("suma");
    Calculadora4 calculu = new Calculadora4(num1,num2);
    int resultado = calculu.suma();
    assertEquals(resul, resultado);
}
```

Watches	Test Results ×	Output - Calculadora4 (test)
calculadora2.Calculadora2Test ×	calculadora4.Calculadora4Test ×	
Tests passed: 66,67 %		
2 tests passed, 1 test failed. (0,112 s)		
calculadora4.Calculadora4Test Failed		
testSuma[0] passed (0,002 s)		
testSuma[1] passed (0,001 s)		
testSuma[2] Failed: expected: <2> but was: <4>		

suma
 suma
 suma

Suite de pruebas

- Permite ejecutar pruebas para todos los métodos en una única clase (no así en las pruebas parametrizadas)
- Test Suites
- New > Other > Unit Test > Test Suites
- Asignar un nombre



Suite de pruebas

- Realizar pruebas parametrizadas para cada una de las operaciones.

The screenshot displays an IDE interface with a project explorer on the left, a code editor in the center, and a test results window at the bottom.

Project Explorer: Shows the project structure with packages `calculadora4` and `test`. The `test` package contains the following classes:

- `Calculadora4TestMultiplica.java`
- `Calculadora4TestResta.java`
- `Calculadora4TestSuite.java`
- `Calculadora4TestSuma.java`

Code Editor: Shows the source code of `Calculadora4TestSuite.java`:

```
6 package calculadora4;
7
8 import org.junit.AfterClass;
9 import org.junit.BeforeClass;
10 import org.junit.runner.RunWith;
11 import org.junit.runners.Suite;
12
13 /**
14  *
15  * @author mcruz
16  */
17 @RunWith(Suite.class)
18 @Suite.SuiteClasses({
19     Calculadora4TestMultiplica.class,
20     Calculadora4TestSuma.class,
21     Calculadora4TestResta.class})
22 public class Calculadora4TestSuite {
23
24 }
```

Test Results Window: Shows the execution results of the test suite. The top bar indicates "Tests passed: 100,00 %". The main area lists the results of 9 tests:

- All 9 tests passed. (0,099 s)
- calculadora4.Calculadora4TestSuite passed
- calculadora4.Calculadora4TestMultiplica.testMultiplica[0] passed (0,003 s)
- calculadora4.Calculadora4TestMultiplica.testMultiplica[1] passed (0,0 s)
- calculadora4.Calculadora4TestMultiplica.testMultiplica[2] passed (0,001 s)
- calculadora4.Calculadora4TestSuma.testSuma[0] passed (0,0 s)
- calculadora4.Calculadora4TestSuma.testSuma[1] passed (0,0 s)
- calculadora4.Calculadora4TestSuma.testSuma[2] passed (0,001 s)
- calculadora4.Calculadora4TestResta.testResta[0] passed (0,0 s)

Test Results Table:

Test Class	Test Method	Result	Duration
calculadora4.Calculadora4TestSuite		passed	
calculadora4.Calculadora4TestMultiplica	testMultiplica[0]	passed	0,003 s
calculadora4.Calculadora4TestMultiplica	testMultiplica[1]	passed	0,0 s
calculadora4.Calculadora4TestMultiplica	testMultiplica[2]	passed	0,001 s
calculadora4.Calculadora4TestSuma	testSuma[0]	passed	0,0 s
calculadora4.Calculadora4TestSuma	testSuma[1]	passed	0,0 s
calculadora4.Calculadora4TestSuma	testSuma[2]	passed	0,001 s
calculadora4.Calculadora4TestResta	testResta[0]	passed	0,0 s

Junit y excepciones

- En **JUnit**, hay tres formas de testear las excepciones:
 - **@Test**, con el atributo 'expected' opcional
 - Try-catch y fail()
 - **@Rule ExpectedException**

1. @Test expected attribute

Uso: cuando quieres testar únicamente el tipo de excepción indicada en el atributo

```
public class Exception1Test {  
    @Test(expected = ArithmeticException.class)  
    public void testDivisionWithException() {  
        int i = 1 / 0;  
    }  
  
    @Test(expected = IndexOutOfBoundsException.class)  
    public void testEmptyList() {  
        new ArrayList<>().get(0);  
    }  
}
```

```
public class Exception2Test {
```

```
    @Test
```

```
    public void testDivisionWithException() {
```

```
        try {
```

```
            int i = 1 / 0;
```

```
            fail(); //recuerda esta línea, de lo contrario puede ser falso positivo
```

```
        } catch (ArithmeticException e) {
```

```
            assertThat(e.getMessage(), is("/ by zero"));
```

```
        }
```

```
    }
```

```
    @Test
```

```
    public void testEmptyList() {
```

```
        try {
```

```
            new ArrayList<>().get(0);
```

```
            fail();
```

```
        } catch (IndexOutOfBoundsException e) {
```

```
            assertThat(e.getMessage(), is("Index: 0, Size: 0"));
```

```
        }
```

```
    }
```

```
}
```

2. Try-catch y fail(): usado en JUnit 3. Para probar el tipo de excepción y también sus instrucciones asociadas

```

public class Exception3Test {
    @Rule
    public ExpectedException thrown = ExpectedException.none();
    @Test
    public void testDivisionWithException() { thrown.expect(ArithmeticException.class);
        thrown.expectMessage(containsString("/ by zero"));
        int i = 1 / 0;
    }
    @Test
    public void testNameNotFoundException() throws NameNotFoundException { //test
type
        thrown.expect(NameNotFoundException.class);
        //test message
        thrown.expectMessage(is("Name is empty!"));
        //test detail
        thrown.expect(hasProperty("errCode")); //make sure getters n setters are defined.
        thrown.expect(hasProperty("errCode", is(666)));
        CustomerService cust = new CustomerService();
        cust.findByName("");
    }
}

```

3. @Rule ExpectedException: Esta regla (a partir de JUnit 4.7) permite probar tanto la excepción como el contenido de la excepción. Pareceido a @Test y fail() pero más elegante

Exception3Test.java

```
public class NameNotFoundException extends Exception {  
    private int errCode;  
    public NameNotFoundException(int errCode, String message) {  
        super(message);  
        this.errCode = errCode;  
    }  
    public int getErrCode() {  
        return errCode;  
    }  
    public void setErrCode(int errCode) {  
        this.errCode = errCode;  
    }  
}
```

NameNotFoundException.java

```
package com.mkyong.examples;
import com.mkyong.examples.exception.NameNotFoundException;
public class CustomerService {
    public Customer findByName(String name) throws
        NameNotFoundException {
        if ("".equals(name)) {
            throw new NameNotFoundException(666, "Name is empty!");
        }
        return new Customer(name);
    }
}
```

CustomerService.java

10. Documentación de la prueba

Documentación de la prueba

- Pruebas bien documentadas → base de futuras tareas de comprobación.
- **Métrica v.3**, proponen que la documentación de la fase de pruebas se basen en los estándares ANSI/IEEE sobre verificación y validación de software.
- Documentos
 - **Plan de Pruebas:** planificación general. Se inicia el proceso de Análisis del Sistema.
 - **Especificación del diseño de pruebas.** Ampliación y detalle del plan de pruebas.
 - **Especificación de un caso de prueba.** A partir de la especificación del diseño de pruebas.
 - **Especificación de procedimiento de prueba.** Detalle del modo en que van a ser ejecutados cada uno de los casos de prueba.
 - **Registro de pruebas.** Registros de los sucesos que tienen lugar durante las pruebas.
 - **Informe de incidente de pruebas.** Incidente, defecto detectado, solicitud de mejora, etc.
 - **Informe sumario de pruebas.** Resumen de las actividades de prueba.