

# ENTORNOS DE DESARROLLO

**IES Santiago Hernández**  
**Curso 2017-2018**  
**Ignacio Agudo Sancho**

# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD

# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD

# Introducción

- Las pruebas de software consisten en verificar y validar un producto software antes de su puesta en marcha.
- Consiste en probar la aplicación construida.
- Se integran dentro de la vida del software.
- Se han considerado como una etapa dentro del desarrollo de software aunque la tendencia está cambiando puesto que se integran en la etapa de desarrollo.

# Introducción

- La ejecución de pruebas involucra una serie de etapas:
  - Planificación de las pruebas
  - Diseño y construcción de los casos de prueba
  - Definición de los procedimientos de prueba
  - Ejecución de las pruebas
  - Registro de resultados obtenidos
  - Registro de errores encontrados
  - Depuración de los errores
  - Informe de los resultados obtenidos

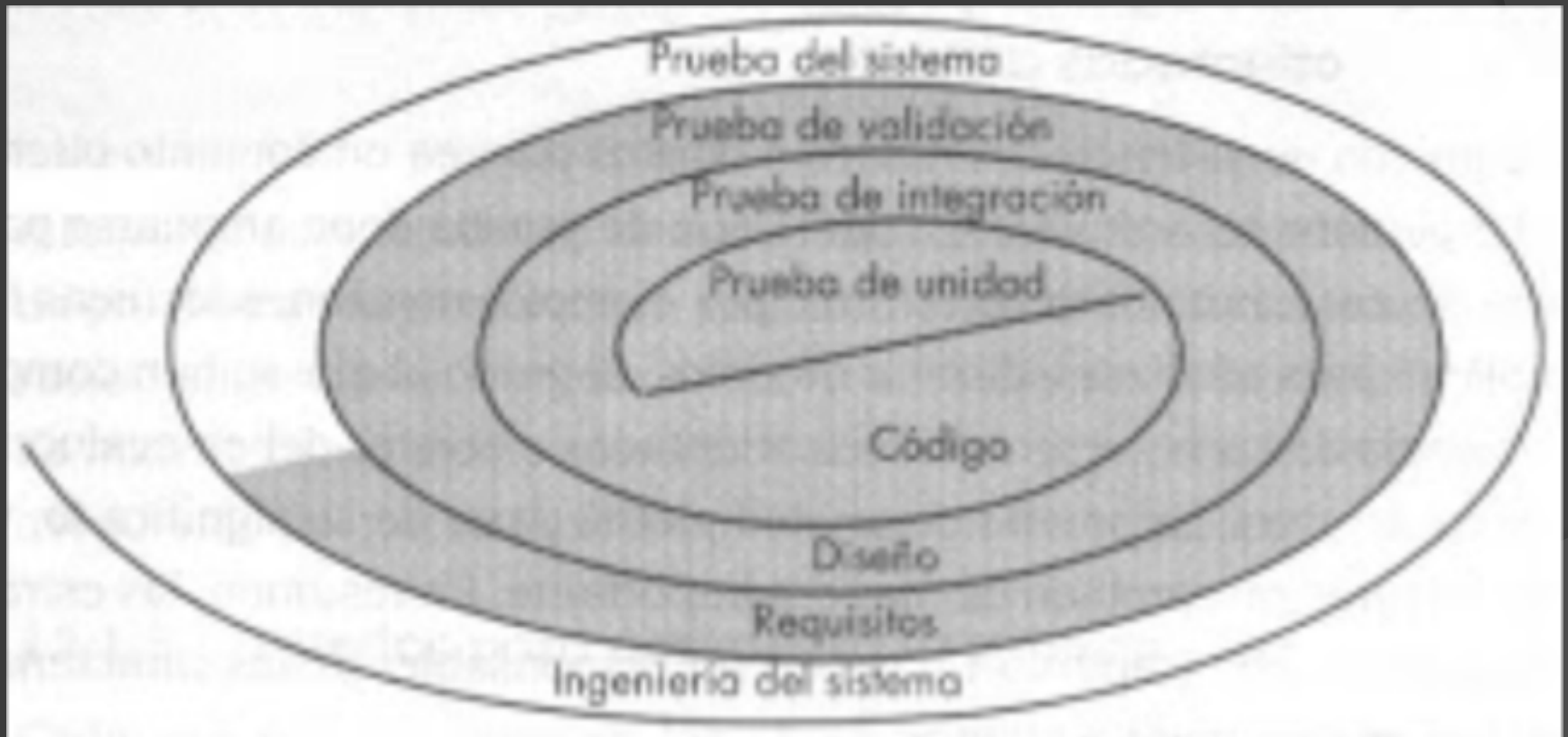
# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD

# Estrategias de Pruebas de Software

- Pruebas de unidad: se centran en la unidad más pequeña de software, el módulo.
- Pruebas de integración: se construye con los módulos una estructura de programa tal como dicta el diseño.
- Prueba de validación o aceptación: prueba del software en el entorno real de trabajo por el usuario final. Se validan los requisitos establecidos.
- Prueba del sistema: verifica que cada elemento encaja de forma adecuada y se alcanza la funcionalidad y rendimiento total.

# Estrategias de Pruebas de Software





# Estrategias de Pruebas de Software

## ● PRUEBA DE UNIDAD (unitaria)

- Se trata de probar cada módulo para eliminar errores en la interfaz y en la lógica interna.
- Utiliza técnicas de caja blanca y caja negra.
- Se realizan pruebas sobre:
  - La interfaz del módulo, para asegurar el flujo correcto.
  - Las estructuras de datos locales, para asegurar que mantienen la integridad.
  - Las condiciones límite.
  - Todos los caminos independientes, para asegurar que todas sentencias se ejecutan al menos una vez.
  - Todos los caminos de manejo de errores.

# 7. Pruebas Unitarias y TDD

- Introducción
- Técnicas de diseño de casos de prueba
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD

# Pruebas de código

- Consiste en ejecutar el programa o parte de él con el objetivo de encontrar errores.
- Se parte de un conjunto de entradas y una serie de condiciones de ejecución.
- Se observan y registran los resultados y se comparan con los resultados esperados.
- Se observará si el comportamiento del programa es el previsto o no y por qué.
- A continuación vemos las pruebas unitarias con JUnit:

# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD

# Pruebas unitarias con JUnit

- Es una herramienta para realizar pruebas unitarias automatizadas.
- Integrada en Eclipse
- Las pruebas unitarias se realizan sobre una clase para probar su comportamiento de modo aislado.
- Veamos un ejemplo. Vamos a crear una clase de prueba llamada Calculadora:

# Pruebas unitarias con JUnit

```
Calculadora.java
1
2 public class Calculadora {
3     private int num1, num2;
4
5     public Calculadora(int a, int b){
6         num1 = a;
7         num2 = b;
8     }
9
10    public int suma(){
11        int resul = num1 + num2;
12        return resul;
13    }
14
15    public int resta(){
16        int resul = num1 - num2;
17        return resul;
18    }
19
20    public int multiplica(){
21        int resul = num1 * num2;
22        return resul;
23    }
24
25    public int divide(){
26        int resul = num1 / num2;
27        return resul;
28    }
29 }
```

# Pruebas unitarias con JUnit

- Una vez tenemos la clase creada, hay que crear la clase de prueba. Para ello:
  - Seleccionamos la clase Calculadora
  - Pulsamos el botón derecho del ratón
  - New → JUnit Test Case
- En la siguiente ventana que se nos abrirá:
  - Seleccionaremos New JUnit 4 test y dejamos el resto de valores por defecto.
  - En el nombre de la clase por defecto pondrá el nombre de la clase original seguido de “Test”, en nuestro caso “CalculadoraTest”.

# Pruebas unitarias con JUnit

**New JUnit Test Case**

The use of the default package is discouraged.

☐ New JUnit 3 test ☒ New JUnit 4 test

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()  
☐ setUp() ☐ tearDown()  
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

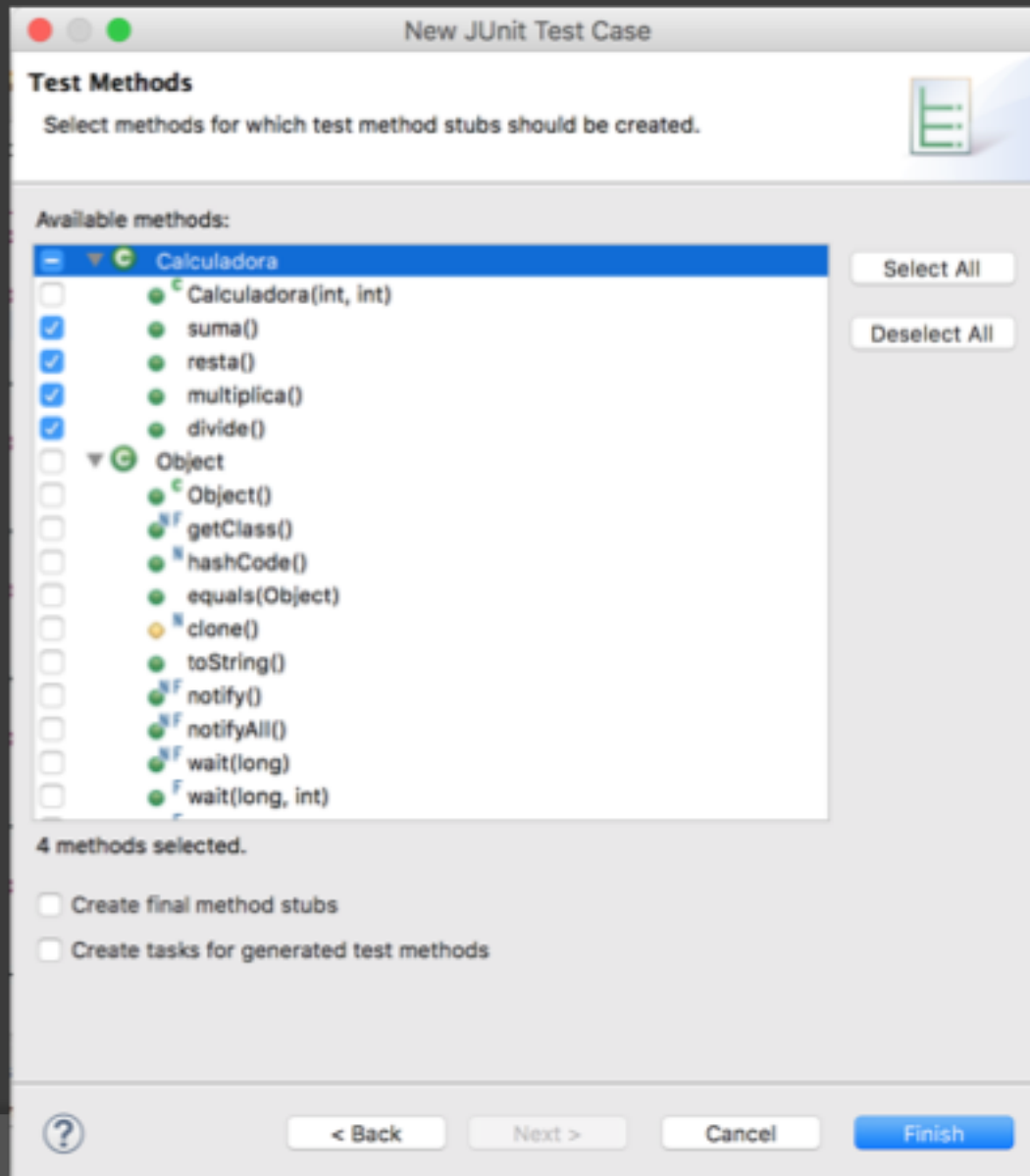
Class under test:



# Pruebas unitarias con JUnit

- Pulsaremos NEXT, y deberemos seleccionar los métodos de nuestra clase que queremos probar.
  - Seleccionaremos los cuatro métodos
  - y pulsaremos en FINISH.

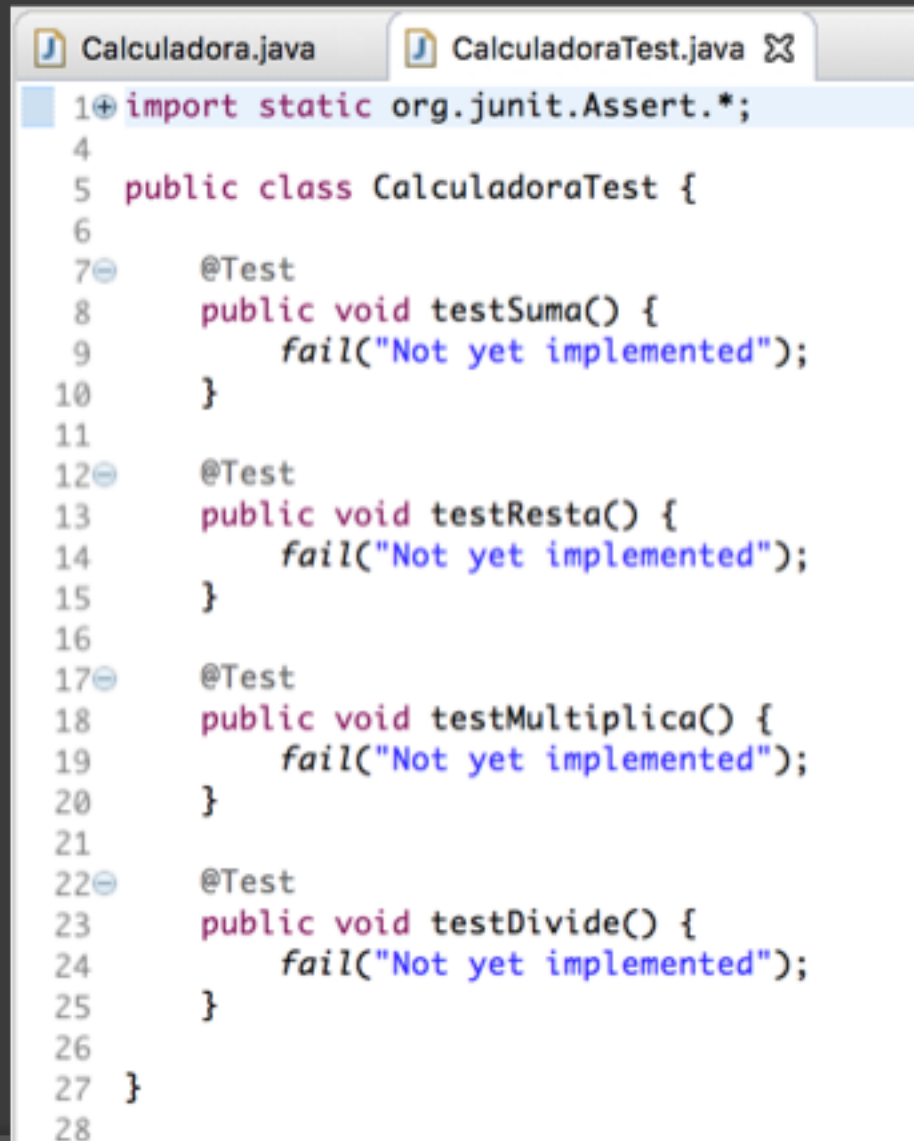
# Pruebas unitarias con JUnit



# Pruebas unitarias con JUnit

- La clase de prueba se creará automáticamente.
- Podremos observar lo siguiente:
  - Se crea un método de prueba para cada método seleccionado anteriormente.
  - Los métodos son públicos, y no devuelven ni reciben.
  - El nombre de los métodos incluye la palabra “test”.
  - Sobre cada método aparece “@Test”, que indica al compilador que es un método de prueba.
  - Cada método de prueba tiene una llamada al método *fail()* con un mensaje indicando que todavía no se ha implementado el método. Este método hace que el test termine con fallo lanzando el mensaje.

# Pruebas unitarias con JUnit



The image shows a code editor with two tabs: 'Calculadora.java' and 'CalculadoraTest.java'. The 'CalculadoraTest.java' tab is active, displaying a Java class with four unit tests. Each test method is annotated with '@Test' and calls 'fail("Not yet implemented");'.

```
1+ import static org.junit.Assert.*;
4
5 public class CalculadoraTest {
6
7     @Test
8     public void testSuma() {
9         fail("Not yet implemented");
10    }
11
12    @Test
13    public void testResta() {
14        fail("Not yet implemented");
15    }
16
17    @Test
18    public void testMultiplica() {
19        fail("Not yet implemented");
20    }
21
22    @Test
23    public void testDivide() {
24        fail("Not yet implemented");
25    }
26
27 }
28
```

# Pruebas unitarias con JUnit

- Debemos conocer algunos métodos propios de JUnit para las comprobaciones:

MÉTODOS	MISIÓN
<code>assertTrue(boolean expresión)</code>	Comprueba que la expresión se evalúe a true.
<code>assertFalse</code>	Comprueba que la expresión se evalúe a false.
<code>assertEquals(valorEsperado, valorReal)</code>	Comprueba que los valores sean iguales.
<code>assertNull(Object objeto)</code>	Comprueba que el objeto sea null.
<code>assertNotNull(Object objeto)</code>	Comprueba que el objeto no sea null.
<code>assertSame(Object objetoEsperado, Object objetoReal)</code>	Comprueba que los objetos sean iguales.
<code>assertNotSame(Object objetoEsperado, Object objetoReal)</code>	Comprueba que los objetos no sean iguales.
<code>fail()</code>	Hace que la prueba falle.

# Pruebas unitarias con JUnit

- Cualquiera de los métodos anteriores puede ser llamado pasándole como parámetro, además, un String que será mostrado en caso de que se produzca error.
- Por ejemplo:
  - `assertTrue(String mensaje, boolean expresión)`
    - Si no es true, al producirse el error se verá el mensaje.
- Veamos un ejemplo: Creamos el test para el método suma y ejecutamos:

# Pruebas unitarias con JUnit

The screenshot displays an IDE interface with two main panels. The left panel shows the 'JUnit' test runner results, and the right panel shows the source code of the test class.

**JUnit Test Results (Left Panel):**

- Package Explorer: JUnit
- Finished after 0,05 seconds
- Runs: 4/4 | Errors: 0 | Failures: 3
- Test Results:
  - CalculadoraTest [Runner: JUnit 4] (0,002 s)
    - testResta (0,001 s)
    - testSuma (0,000 s) **Passed**
    - testMultiplica (0,000 s)
    - testDivide (0,000 s)
- Failure Trace (bottom left)

**Source Code (Right Panel):**

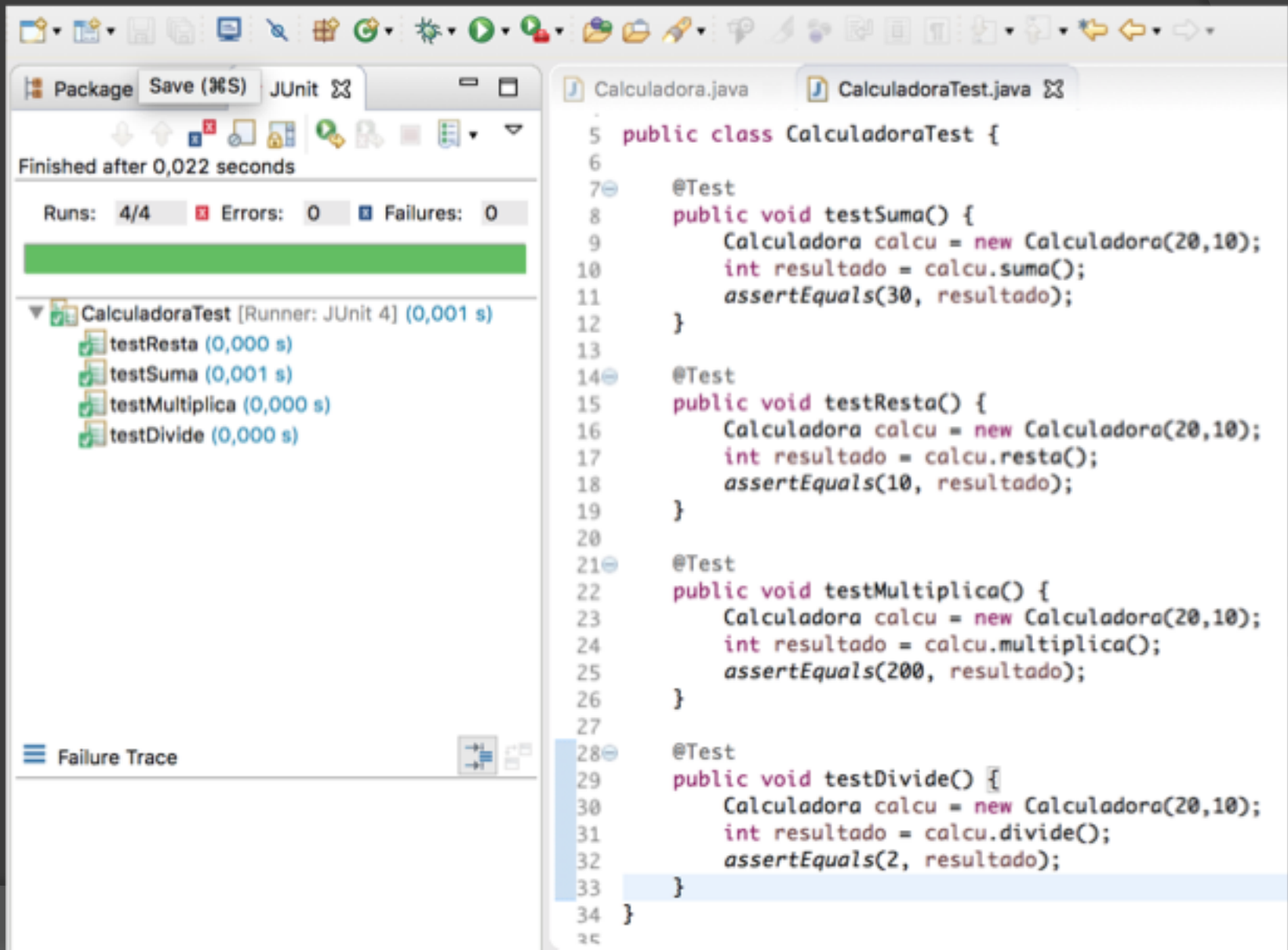
```
Calculadora.java | CalculadoraTest.java
1 import static org.junit.Assert.*;
4
5 public class CalculadoraTest {
6
7     @Test
8     public void testSuma() {
9         Calculadora calculo = new Calculadora(10,20);
10        int resultado = calculo.suma();
11        assertEquals(30, resultado);
12    }
13
14    @Test
15    public void testResta() {
16        fail("Not yet implemented");
17    }
18
19    @Test
20    public void testMultiplica() {
21        fail("Not yet implemented");
22    }
23
24    @Test
25    public void testDivide() {
26        fail("Not yet implemented");
27    }
28
29 }
30
```

# Pruebas unitarias con JUnit

- Al ejecutar (Run) observamos varias cosas:
  - Que ha pasado bien el test de la suma, puesto que sale con un “tic” verde.
  - Que ha fallado en el resto de métodos, puesto que sale una X con fondo azul.
  - Que no ha habido errores, pues éstos saldrían con una X con fondo rojo.
  - Nos indica que ha ejecutado 4/4 métodos de @Test, ha encontrado 0 Errors y 3 Failures.
- Implementamos ahora las pruebas unitarias para el resto de métodos, y ejecutamos de nuevo:



# Pruebas unitarias con JUnit



The screenshot displays an IDE interface with two main panels. The left panel shows the JUnit test runner results, and the right panel shows the source code for the calculator and its tests.

**JUnit Test Results (Left Panel):**

- Package: Save (⌘S) JUnit
- Finished after 0,022 seconds
- Runs: 4/4 Errors: 0 Failures: 0
- Test Results:
  - CalculadoraTest [Runner: JUnit 4] (0,001 s)
    - testResta (0,000 s)
    - testSuma (0,001 s)
    - testMultiplica (0,000 s)
    - testDivide (0,000 s)
- Failure Trace

**Source Code (Right Panel):**

**Calculadora.java**

```
5 public class Calculadora {
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34 }
```

**CalculadoraTest.java**

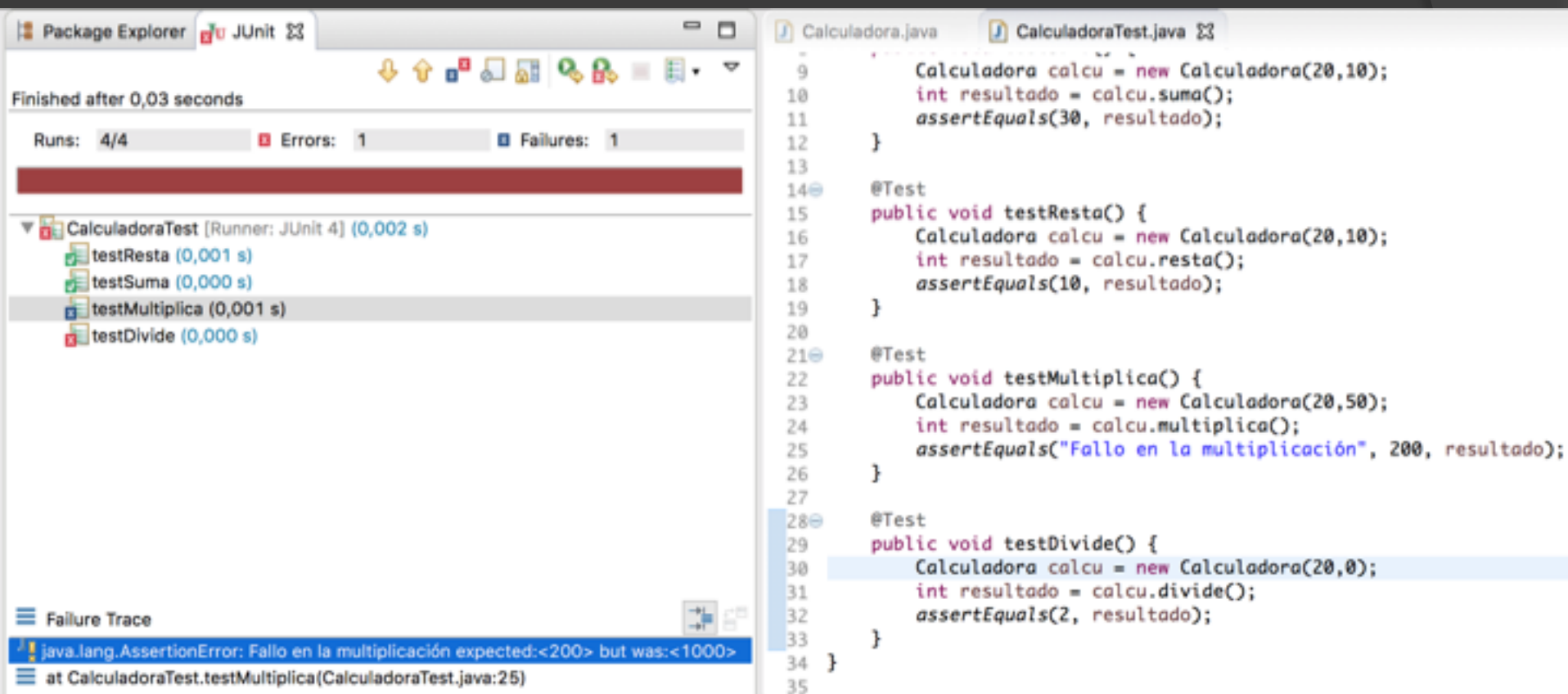
```
5 public class CalculadoraTest {
6
7     @Test
8     public void testSuma() {
9         Calculadora calculo = new Calculadora(20,10);
10        int resultado = calculo.suma();
11        assertEquals(30, resultado);
12    }
13
14     @Test
15     public void testResta() {
16        Calculadora calculo = new Calculadora(20,10);
17        int resultado = calculo.resta();
18        assertEquals(10, resultado);
19    }
20
21     @Test
22     public void testMultiplica() {
23        Calculadora calculo = new Calculadora(20,10);
24        int resultado = calculo.multiplica();
25        assertEquals(200, resultado);
26    }
27
28     @Test
29     public void testDivide() {
30        Calculadora calculo = new Calculadora(20,10);
31        int resultado = calculo.divide();
32        assertEquals(2, resultado);
33    }
34 }
```

# Pruebas unitarias con JUnit

- Para ver la diferencia entre fallo y error, cambiamos el código de dos de los métodos:
  - Para que *multiplica()* falle, hacemos que el valor esperado no coincida con el resultado; además añadimos un String en el método `assertEquals` para que lance el mensaje cuando falle.
  - Para que *divide()* produzca un error, al crear el objeto calculadora asignamos el valor 0 al segundo parámetro (al dividir por 0 se produce una excepción).

# Pruebas unitarias con JUnit

- La traza de ejecución (abajo) nos dice qué ha fallado.



The screenshot displays an IDE with two main panes. The left pane shows the JUnit test results, and the right pane shows the source code of the `Calculadora` class and its tests.

**JUnit Test Results (Left Pane):**

- Package Explorer: JUnit
- Finished after 0,03 seconds
- Runs: 4/4
- Errors: 1
- Failures: 1
- Test Results:
  - CalculadoraTest [Runner: JUnit 4] (0,002 s)
    - testResta (0,001 s)
    - testSuma (0,000 s)
    - testMultiplica (0,001 s)
    - testDivide (0,000 s)
- Failure Trace:
  - java.lang.AssertionError: Fallo en la multiplicación expected:<200> but was:<1000>
  - at CalculadoraTest.testMultiplica(CalculadoraTest.java:25)

**Source Code (Right Pane):**

`Calculadora.java`

```
9     Calculadora calcul = new Calculadora(20,10);
10     int resultado = calcul.suma();
11     assertEquals(30, resultado);
12 }
13
14 @Test
15 public void testResta() {
16     Calculadora calcul = new Calculadora(20,10);
17     int resultado = calcul.resta();
18     assertEquals(10, resultado);
19 }
20
21 @Test
22 public void testMultiplica() {
23     Calculadora calcul = new Calculadora(20,50);
24     int resultado = calcul.multiplica();
25     assertEquals("Fallo en la multiplicación", 200, resultado);
26 }
27
28 @Test
29 public void testDivide() {
30     Calculadora calcul = new Calculadora(20,0);
31     int resultado = calcul.divide();
32     assertEquals(2, resultado);
33 }
34 }
35
```

`CalculadoraTest.java`

# Pruebas unitarias con JUnit

- El siguiente test comprueba que la llamada al método *divide()* devuelve la excepción *ArithmeticException* al dividir 20 entre 0; y por tanto sale por la cláusula *catch*.
- Si no se lanza la excepción, se lanza el método *fail()* con un mensaje.
- La prueba tiene éxito si se produce la excepción, y falla en caso contrario:

```
35 @Test
36 public void testException(){
37     try{
38         Calculadora calculo = new Calculadora(20,0);
39         int resultado = calculo.divide();
40         fail("Fallo, debería haber lanzado la excepción");
41     }catch (ArithmeticException e){
42         //Prueba satisfactoria
43     }
44 }
```

# Pruebas unitarias con JUnit

## ● EJERCICIO

- Partiendo del proyecto de la calculadora, modificar el método `resta()` y añadir los métodos `resta2()` y `divide2()` que tienes a continuación.
- Después crea los tests para probar los tres métodos.
- Utiliza los métodos según convengan:
  - `assertTrue()`
  - `assertFalse()`
  - `assertNull()`
  - `assertNotNull()`
  - `assertEquals()`

# Pruebas unitarias con JUnit

## ● EJERCICIO

- Los métodos para realizar el ejercicio son:

```
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;  
}
```

# Pruebas unitarias con JUnit

## ● EJERCICIOS

- Toma como referencia el pseudocódigo de los ejercicios de camino básico y pásalo a código, de tal manera que esté en métodos para los que se puedan hacer pruebas unitarias sencillas.
- Haz cada uno de ellos en una clase diferente.
- A continuación crea las clases de pruebas de cada uno de ellos e implementa los diferentes casos de prueba que se consiguieron en la realización de los ejercicios de calcular la complejidad ciclomática.

# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD



# TDD

- ⦿ Test Driven Development
- ⦿ Es una forma de trabajo que implica un cambio de mentalidad.
- ⦿ Consiste en codificar pruebas, desarrollar y refactorizar de forma continua el código.
- ⦿ La idea principal es codificar las pruebas unitarias para el código que tenemos que implementar, y luego desarrollar la lógica de negocio.

# TDD: Ventajas

- ⦿ Mayor calidad en el código desarrollado.
- ⦿ Diseño orientado a las necesidades.
- ⦿ Simplicidad, nos enfocamos en el requisito concreto.
- ⦿ Menor redundancia.
- ⦿ Mayor productividad (menor tiempo de debugging).
- ⦿ Reducción del número de errores.

# TDD: Condiciones

- Tener bien definidos los requisitos de la función a realizar.
- Criterios de aceptación, contemplando todos los casos posibles tanto válidos como de error.
- Ceñirnos únicamente en testear la lógica de negocio.
- Cada casuística para cada criterio de aceptación debería llevar su prueba asociada.

# TDD: Ciclo de vida

- Se basa en una continua codificación y refactorización:
  - Elegir un requisito.
  - Codificar la prueba.
  - Verificar que la prueba falla.
    - Si no falla es porque el requerimiento ya estaba implementado o porque la prueba es errónea.
  - Codificar la implementación.
  - Ejecutar las pruebas automatizadas.
  - Refactorizar.
  - Actualizar la lista de requisitos.

# 7. Pruebas Unitarias y TDD

- Introducción
- Estrategias de prueba del software
- Pruebas de código
- Herramientas de depuración
- Pruebas unitarias JUnit
- TDD