Projecte de Programació

JUnit

Prueba de Programas: Java

Tests Unitarios: Junit* (**`https://junit.org/junit4`**)

JUnit es un *framework* que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si cada uno de los métodos de la clase se comporta como se espera

También puede servir para controlar las pruebas de regresión necesarias cuando parte del código ha sido modificado

Integrado dentro de los IDEs más habituales (NetBeans, Eclipse, IntelliJ, etc.)

*Usaremos JUnit 4.12, aunque ya existe JUnit 5

Prueba de Programas: Java

Tests Unitarios: Un test unitario (*unit test*) examina el comportamiento de una unidad de trabajo (*unit of work*) diferenciada. En Java usualmente esta unidad de trabajo es un método (una unidad de trabajo diferenciada es una tarea que no depende de ninguna otra tarea)

Test Class (también llamada ***TestCase***): Una clase que contiene uno o más métodos *anotados** con **`@Test`**. Sirven para agrupar uno o más tests que verifican comportamientos relacionados

Test → método anotado con **`@Test`**

*Para trabajar con JUnit las anotaciones son fundamentales. Con ellas definiremos *suites*, *runners* y métodos que se ejecuten en momentos estratégicos de los tests

Prueba de Programas: Java

Suite (o ***Test Suite***): Grupo de tests. Una *test suite* es una manera conveniente de agrupar tests relacionados. Si no se define ninguna, JUnit crea una por defecto, que incluye todos los tests que hay en la *test class*. Usualmente una *suite* incluye *test classes* del mismo *package*
Está en **`org.junit.TestSuite`**

Runner (o ***Test Runner***): El *runner* es la clase que ejecuta las *test suites*. JUnit proporciona diversos *runners* por defecto, pero también podemos definir propios
Están en **`org.junit.runners.*`**

Prueba de Programas: Java

Veamos un ejemplo sencillo. Tenemos una clase para calcular sumas, restas, productos y divisiones:

```
package main.domini;

public class Calculator {
    public double add(double n1,double n2) {
        return n1 + n2;
    }
    public double multiply(double n1, double n2) {
        return n1 * n2;
    }
    public double subtract(double n1, double n2) {
        return n1 - n2;
    }
    public double divide(double n1, double n2) {
        return n1 / n2;
    }
}
```

Podríamos hacer un test de la siguiente manera...

Prueba de Programas: Java

```
package test.domini;

import main.domini.*;
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void add() {
        Calculator calculator = new Calculator();
        double result = calculator.add( 10, 50 );
        assertEquals( 60, result, 0 );
    }

    @Test
    public void multiply() { . . . }
    @Test
    public void subtract() { . . . }
    @Test
    public void divide() { . . . }
}
```

Prueba de Programas: Java

```
n00ne src $
n00ne src $ echo $CLASSPATH
.:/home/n00ne/opt/java/junit-4.12.jar:/home/n00ne/opt/java/hamcrest-core-1.3.jar
n00ne src $
n00ne src $
n00ne src $ javac main/domini/Calculator.java
n00ne src $
n00ne src $
n00ne src $ javac test/domini/CalculatorTest.java
n00ne src $
n00ne src $
n00ne src $ java org.junit.runner.JUnitCore test.domini.CalculatorTest
JUnit version 4.12
....
Time: 0.004

OK (4 tests)
```

Prueba de Programas: Java

Si hacemos que falle un test, por ejemplo el *add*...

```
package test.domini;

import src.domini.*;
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {
    @Test
    public void add() {
        Calculator calculator = new Calculator();
        double result = calculator.add( 10, 50 );
        assertEquals( 40, result, 0 );
    }
    . . .
}
```

```
n00ne src $ java org.junit.runner.JUnitCore test.domini.CalculatorTest
JUnit version 4.12
...E.
Time: 0.01
There was 1 failure:
1) add(test.domini.CalculatorTest)
java.lang.AssertionError: expected:<40.0> but was:<60.0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:834)
        at org.junit.Assert.assertEquals(Assert.java:553)
        at org.junit.Assert.assertEquals(Assert.java:683)
        at test.domini.CalculatorTest.add(CalculatorTest.java:42)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
        at org.junit.runners.Suite.runChild(Suite.java:128)
        at org.junit.runners.Suite.runChild(Suite.java:27)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:363)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:137)
        at org.junit.runner.JUnitCore.run(JUnitCore.java:115)
        at org.junit.runner.JUnitCore.runMain(JUnitCore.java:77)
        at org.junit.runner.JUnitCore.main(JUnitCore.java:36)

FAILURES!!!
Tests run: 4,  Failures: 1
```

Prueba de Programas: Java

En los ejemplos hemos usado **`org.junit.runner.JUnitCore`** , pero podríamos usar un **`TestRunner`** propio:

```
package test.domini;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunnerPropio {

    public static void main(String[] args) {
      Result result = JUnitCore.runClasses(CalculatorTest.class);
      for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
      }
      if (result.wasSuccessful()) {
            System.out.println("El test ha ido bien");
      } else {
            System.out.println("El test ha ido mal");
      }
    }
}
```

Prueba de Programas: Java

La ejecución en este caso sería de la siguiente forma:

```
src $ javac test/domini/CalculatorTest.java

src $ javac test/domini/TestRunnerPropio.java

src $ java test.domini.TestRunnerPropio
El test ha ido bien
src $
```

Prueba de Programas: Java

Una de les clases más importantes de JUnit es **`org.junit.Assert`** (subclase de **`Object`**). Contiene los métodos que nos permiten verificar relaciones entre los resultados reales y los resultados esperados. Algunos ejemplos son los siguientes:

- **`assertEquals(…)`** / **`assertNotEquals(…)`**
- **`assertNull(…)`** / **`assertNotNull(…)`**
- **`assertSame(…)`** / **`assertNotSame(…)`**
- **`assertTrue(…)`** / **`assertFalse(…)`**
- **`assertThat(…)`**
- **`fail(…)`**

(ver **`http://junit.sourceforge.net/javadoc/org/junit/Assert.html`**)

Prueba de Programas: Java

Podemos detallar el funcionamiento de algunos **`asserts`**:

- **`assertEquals("message", A, B)`**
  Afirma que **`A`** y **`B`** son iguales utilitzando **`A.equals(B)`**

- **`assertNotNull("message", A)`**
  Afirma que el objeto **`A`** no es **null**

- **`assertSame("message", A, B)`**
  Afirma que **`A`** y **`B`** son el mismo objeto, usando **`A == B`**
  (*aliasing*)

- **`assertTrue("message", A)`**
  Afirma que la condición booleana **`A`** es **`true.`**

Prueba de Programas: Java

La anotación **@Test** puede tener parámetros opcionales, para poder controlar algunos aspectos de los tests:

- Si queremos controlar el *tiempo* que tarda un método en ejecutarse:

```
@Test(timeout=<tiempo en milisegundos>)
```

- Si queremos verificar que el método fallará, generando una excepción de clase **SomeException** (subclase de **Exception**).

```
@Test(expected=SomeException.class)
```

```
@Test (expected=IdexOutOfBoundsException.class)
public void empty() {
      new ArrayList<Object>().get(0);
}
```

Prueba de Programas: Java

Ya conocemos la anotación **`@Test`**, pero para poder configurar el contexto en el que se ejecutan los tests podemos usar otras anotaciones:

- **`@Before`**: Método que se ejecutará *antes* de cada test, sin importar si el test falla o no. Ha de ser público

- **`@After`**: Método que se ejecutará *después* de cada test, sin importar si el test falla o no. Ha de ser público

Si en una misma clase hay diversos métodos anotados con **`@Before`** y **`@After`** hay que tener en cuenta que el orden de ejecución es indefinido

- **`@BeforeClass`**: Método que se ejecutará una vez *antes* de cualquier test de la clase. Ha de ser público y *static*

- **`@AfterClass`**: Método que se ejecutará una vez *después* de cualquier test de la clase. Ha de ser público y *static*

Prueba de Programas: Java

Ya conocemos la anotación **`@Test`**, pero para poder configurar el contexto en el que se ejecutan los tests podemos usar otras anotaciones:

- **`@Ignore`**: Método que será ignorado en cada ejecución de la clase de tests

Hay que tener en cuenta que JUnit instanciará la *test class* antes de invocar cada test (recordemos, método anotado con **`@Test`**). Aún así, los métodos anotados con **`@BeforeClass`** y **`@AfterClass`** se ejecutarán sólo una vez, antes de la primera instanciación de la clase y cuando acaben todos los tests, respectivamente

Prueba de Programas: Java

```
package test.domini;
import org.junit.*;
public class EjemploAnotaciones {
    @BeforeClass
    public static void beforeClass() {
      System.out.println("Antes de instanciar por primera vez la clase");
    }
    @AfterClass
    public static void afterClass() {
        System.out.println("Después de acabar todos los tests");
    }
    @Before
    public void before() {
       System.out.println("Antes de test");
    }
    @After
    public void after() {
       System.out.println("Después de test");
    }
    @Test
    public void test1() {
       System.out.println("--> test 1");
    }
    @Test
    public void test2() {
       System.out.println("--> test 2");
    }
    @Ignore
    @Test
    public void ignoreTest() {
        System.out.println("test ignorado");
    }
}
```

```
$ java org.junit.runner.JUnitCore
       test.domini.EjemploAnotaciones

JUnit version 4.12

Antes de instanciar por primera vez la clase
Antes de test
--> test 1
Después de test
Antes de test
--> test 2
Después de test
Después de acabar todos los tests

Time: 0,015

OK (2 tests)
```

Prueba de Programas: Java

La idea es que los tests generalmente sigan el esquema:

1.- Preparamos el test creando un entorno controlado con un estado conocido (crearemos los objetos necesarios, adquiriremos los recursos -ficheros, conexiones, etc.- necesarios, etc.). Este estado de *pre-test* se conoce como *test fixture*. Lo haremos con métodos anotados con **`@BeforeClass`**, etc.

2.- Invocamos el método que estamos comprobando, es decir, el test (método con anotación **`@Test`**, eventualmente con parámetros **`expected`** o **`timeout`**)

3.- Confirmamos el resultado, usualmente invocando uno o más métodos *assert*

Opcionalmente, se puede combinar con herramientas para medir *code coverage* como **Jacoco**

Prueba de Programas: Java

Lo más razonable es agrupar los tests en función de las clases que estamos probando. Aún así querremos ejecutarlos todos de una vez, sin tener que ir haciendo las diversas pruebas por cada grupo de clases. Para agrupar tests podemos utilizar las *Suites*

Definiremos *suites* utilizando las anotaciones **@RunWith** y **@SuiteClasses**, convenientemente parametrizadas. Por ejemplo, supongamos que tenemos dos tests:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestCaseA {

  @Test
  public void testA1() {
     assertEquals("test tonto 1", 1+1, 2);
  }

}
```

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestCaseB {

  @Test
  public void testB1() {
     assertTrue("test tonto 2", true);
  }

}
```

Prueba de Programas: Java

Cada uno de estos tests podría ir a una *suite* propia, es decir, que contenga sólo ese test:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(value = Suite.class)
@SuiteClasses(value = { TestCaseA.class })
public class TestSuiteA {}
```

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(value = Suite.class)
@SuiteClasses(value = { TestCaseB.class })
public class TestSuiteB {}
```

Cada una de estas *suites* se podría ejecutar independientemente:

```
$ java org.junit.runner.JUnitCore TestSuiteA (o TestSuiteB)
JUnit version 4.12
.
Time: 0,012

OK (1 test)
```

Fijémonos que las clases **TestSuiteA** y **TestSuiteB** están *vacías*

Prueba de Programas: Java

Finalmente, podríamos *combinar* estas *suites* en otra *suite* que las ejecute :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(value = Suite.class)
@SuiteClasses(value = { TestSuiteA.class, TestSuiteB.class })
public class MasterTestSuite {}
```

```
$ java org.junit.runner.JUnitCore MasterTestSuite
JUnit version 4.12
.
Time: 0,006

OK (2 test)
```

Fijémonos que la clase **`MasterTestSuite`** está *vacía*
(alternativamente, podríamos haber puesto directamente **`TestCaseA`** y **`TestCaseB`** en una *Suite*)

Prueba de Programas: Java

Desde Junit 4, podemos hacer tests **parametrizados**, usando el *test runner* **`org.junit.runners.Parameterized`**. Los tests parametrizados permiten ejecutar el mismo test una y otra vez usando diferentes valores. Se han de seguir 5 pasos para crear un test parametrizado:

1) Anotar la test class con **`@RunWith(Parameterized.class)`**
2) Crear un método *public static* anotado con **`@Parameters`** que retorne una *Collection of Objects* (como *Array*) como conjunto de datos de test
3) Crear una constructora *public* que recibe como entrada el equivalente a una "fila" de datos de test
4) Crear un atributo privado para cada "columna" de datos de test
5) Crear tus juegos de pruebas (test cases) usando estos atributos como origen de los datos de test

El test se invocará una vez por cada fila de datos

Prueba de Programas: Java

En el ejemplo de la clase *Calculator* anterior, supongamos un test para la suma:

```
package test.domini;

import main.domini.*;

import java.util.*;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(value=Parameterized.class)
public class TestingAdd {
    private double expected, valueOne, valueTwo;
    public TestingAdd(double e, double v1, double v2) {
        this.expected = e; this.valueOne = v1; this.valueTwo = v2;
    }

        // continue...
```

Prueba de Programas: Java

Aunque solo haya un método con **@Test**, el test se hará 5 veces, una por cada conjunto de valores definidos en **getTestParameters**:

```
        // . . .

    @Parameters
    public static Collection<Double[]> getTestParameters () {
     return Arrays.asList( new Double[][] {
      { 0.0, 1.0, -1.0},  // expected, valueOne, valueTwo
      { 10.0, 90.0, -80.0},
      { 100.0, -1000.0, 1100.0},
      { 5.0, 2.0, 3.0},
      { 2.0, 1.0, 1.0} });
    }

    @Test
    public void add() {
     Calculator calculator = new Calculator();
     assertEquals( expected, calculator.add(valueOne,valueTwo), 0 );
    }
}
```

Prueba de Programas: Java

Podemos hacer tests similares para las otras operaciones de *Calculator*

Podemos tener:

`TestingMultiply.java`,
`TestingSubtract.java`,
`TestingDivide.java`

Podemos ejecutarlos individualmente, para evaluar cada operación por separado. Por ejemplo:

```
$ java org.junit.runner.JUnitCore test.domini.TestingAdd
JUnit version 4.12
.....
Time: 0,018

OK (5 tests)
```

Prueba de Programas: Java

Pero si queremos hacer un test completo de toda la clase, con todas las operaciones: **`TestingAdd`**, **`TestingMultiply`**, **`TestingSubtract`** i **`TestingDivide`**, podemos utilizar una *suite* para agruparlos todos:

```
package test.domini;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(value = Suite.class)
@SuiteClasses(value = { TestingAdd.class,
                        TestingMultiply.class,
                        TestingSubtract.class,
                        TestingDivide.class })
public class TestingAll {}
```

```
$ java org.junit.runner.JUnitCore test.domini.TestingAll
JUnit version 4.12
....................
Time: 0,015

OK (20 tests)
```

Prueba de Programas: Java

Hamcrest matchers

(**http://junit.org/junit4/javadoc/4.12/org/hamcrest/CoreMatchers.html**)
Utilidad para añadir predicados o restricciones a los valores especificados en los *asserts*, pudiendo especificar declarativamente reglas que se han de cumplir. Hay muchos (ver el *link*):

allOf	**anything**	**is**
anyOf	**not**	**instanceOf**
sameInstance	**nullValue**	**hasItem**
closeTo	**hasEntry**	**equalTo**
hasKey	**hasValue**	**equalToIgnoringCase**

Además, podemos crear nuestros propios *matchers*. Por ejemplo:

```
assertThat(values, hasItem(anyOf(equalTo("one"),
                                 equalTo("two"),
                                 equalTo("three"))));
assertThat("test", anyOf(is("testing"),
                         containString("est")));
```

Prueba de Programas: Java

Mocks & Stubs:

En las pruebas unitarias, frecuentemente necesitamos clases u objetos ajenos a los métodos que estamos comprobando -> Podemos usar *stubs* o *mocks*, hechos por nosotros o por medio de *frameworks**

[Method] Stub: Trozo de código que se inserta en tiempo de ejecución en lugar del código real, para aislar el objeto que lo utiliza de la implementación real. Pretendemos sustituir comportamiento potencialmente complejo por un comportamiento trivial o sencillo que permita el *testing* independiente

Mock objects: Sustitutos de objetos con los que los métodos test interactúan. Donde quiera que se usen en un test, se define su comportamiento esperado en ese punto concreto:

when (ObjetoMockeado.operación(parámetros)).thenReturn (resultado)

Así como puede ser necesario que los *stubs* implementen fragmentos de la lógica de un programa, no es el caso de los *mocks*. Son objetos con métodos esencialmente vacíos

*`http://jmock.org` `http://easymock.org` `https://site.mockito.org`

Prueba de Programas: Java

Mocks & Stubs

Mockito está pensada explícitamente para unit tests en Java. Un ejemplo de uso*:

```
imports ….
@ExtendWith(MockitoExtension.class)
class ServiceDatabaseIdTest {

    @Mock
    Database databaseMock;

    @Test
    void testQuery() {

        assertNotNull (databaseMock);
        when(databaseMock.isAvailable()).thenReturn(true);

        Service t = new Service(databaseMock);
        boolean check = t.query(“* from t”);
        assertTrue(check);
    }
}
```

*https://www.vogella.com/tutorials/Mockito/article.html

Prueba de Programas: Java

Mocks & Stubs

Si para el mismo ejemplo quisiéramos usar un *stub*, tendríamos que construirnos una clase trivial:

```
public class Database {

    ….

    public boolean isAvailable() {
      return true;
    }
}
```

Pero si este mismo *stub* se tuviera que usar en otros tests que esperaran comportamientos diferentes, el código del *stub* se tendría que ir complicando según la casuística

En PROP se puede usar cualquiera de los dos, es decisión de cada miembro del equipo cuando diseñe sus tests

Prueba de Programas: Java

Mocks & Stubs:

A nivel puramente operativo, consideraremos que los *stubs* se han de programar a mano y para los *mocks* se pueden usar herramientas, con lo cual:

Concepto	Pros	Contras
Stub	El concepto es fácil de entender y de implementar.	Cuanta más casuística necesita cubrir el *stub,* más complejo es de implementar. Es código que se va a tirar una vez la clase esté implementada.
Mock	Necesita menos código para hacer lo mismo que hace el *stub*. Cuanta más casuística, más es la distancia del código necesario a favor del *mock*.	La curva de aprendizaje del concepto es más alta que en el caso del *stub*.