JUnit

Unha Introducción aos tests de Unidade con JUnit

Sumario

Introducción

JUnit é un framework para a construcción de test Unitarios en Java. Este framework fai uso de anotacións Java (annotations) para identificar os métodos que forman parte dos tests.

Un ***test JUnit*** é un método contido nunha clase que se utiliza únicamente para probas. Esta clase chámase "Clase para Testing". Para indicar que un método é un método de "test", se indica mediante a *annotation* **@Test**.

Dentro de estes métodos utilizamos o método ***assert*** proporcionado por JUnit ou outro framework, para comprobar que os resultados devoltos polo método que estamos testando se corresponden co resultado esperado. En estas chamadas a ***assert*** deberíamos indicar mensaxes explicativos que permitan identificar correctamente o lugar e motivo do fallo, facilitando así a súa corrección, sobre todo, tendo en conta que o programador encargado de realizar os tests unitarios non é necesariamente o programador que a codificou.

Exemplo de Test Unitario

O seguinte exemplo mostra un test unitario JUnit para a clase MyClass, comprobando o correcto funcionamento do seu método *multiply(int,int);*

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

public class MyTests {
   @Test
   public void multiplicationOfZeroIntegersShouldReturnZero() {
      MyClass tester = new MyClass(); // MyClass is tested

      // assert statements
      assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
      assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
      assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
   }
}
```

Regras de estilo JUnit

Os tests JUnit utilizan distintas convencións de estilo. Un exemplo típico e utilizar como sufixo ou prefixo a palabra Test, para os nomes das clases, como TestCliente ou ClienteTest.

Como regra xeral, o nome dun test debería ser explicativo, indicando qué estamos probando. Si se realiza esto correctamente non é necesario leer o código para saber qué é o que proba o test unitario.

Outra convención común é utilizar "should" nos nomes dos métodos de proba. Por exemplo, *orderClienteShouldBeCreated*, o que indicaría o que debería pasar si o método se executa correctamente.

Outra aproximación común é o uso de Given[explicacion da entrada]When[explicacion do que se fai]Then[resultadoesperado]. Por exemplo, *givenProductoWhenNotExistenciasThenThrowOutofStockException*

A idea xeral é que mediante a creación de identificadores significativos é posible saber perfectamente qué estamos probando sen necesidade de leer o código das probas.

Execución de Test Unitarios JUnit desde o terminal

A clase ***org.junit.runner.JUnitCore*** nos proporciona o método ***runClasses()***. Este método nos permite executar unha ou varias clases de Test JUnit. O valor devolto por este método é un obxecto de tipo ***org.junit.runner.Result***, que podemos utilizar para obter a información sobre os resultados dos tests.

O seguinte exemplo indica como levar a cabo o test unitario JUnit correspondente a clase *MyClass* (*MyClassTest*), indicando na consola os posibles fallos:

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

public class MyTestRunner {
	public static void main(String[] args) {
		Result result = JUnitCore.runClasses(MyClassTest.class);
		for (Failure failure : result.getFailures()) {
			System.out.println(failure.toString());
		}
	}
}
```

Esta clase pode executarse como calqueira outro programa Java dende a liña de comandos, aínda que pode ser necesario engadir a CLASSPATH a localización da librería JUnit.

JUnit supon que todos os métodos de test poden executarse nun orden arbitrario. O código de test ben escrito non debe asumir ningún orden, e decir un test non debe depender de outros. O comportamento por defecto de *JUnit 4* é utilizar un orden non predecible para a execución dos tests.

JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests. E posible utilizar unha *annotation* para definir que os métodos se ordenen lexicográficamente polo seu nome, ou outras variacións:

@FixMethodOrder(MethodSorters.NAME_ASCENDING)

@FixMethodOrder(MethodSorters.DEFAULT)

@FixMethodOrder(MethodSorters.JVM)

Anotacións de Tests JUnit 4

JUnit utiliza anotacións para marcar os métodos como métodos de test e configuralos. A seguinte táboa amosa as anotacións máis importantes para utilizar nos métodos en JUnit 4, que deben ser importadas mediante ***import org.junit.<Anotacion>***

Anotación	Descripción
@Test	Identifica o método como un método de Test JUnit
@Before	Este método se executa antes de cada Test
@After	Este método se executa á finalización de cada test
@BeforeClass	Se executa unha vez ao comenzo do Test. O método debe ser static
@AfterClass	Se executa unha vez ao final do Test. O método debe ser static
@Ignore ou @Ignore("Mensaxe")	Indica que o test non está activo.
@Test (expected = Exception.class)	Falla si o método non lanza a Exception indicada.
@Test(timeout=100)	Falla si o métdo tarda máis de 100ms en executarse

Métodos para Test de JUnit

JUnit nos proporciona a clase **Assert**, que dispón de varios métodos *static* para comprobar certas condicións de execución. Típicamente estes métodos comenzan con ***assert***, permitindo especificar unha mensaxe de erro, o resultado esperado e o resultado actual. O método compara o valor actual co valor esperado, lanzando unha excepción si falla. A seguinte táboa amosa un resumo de estes métodos. Os parámetros entre corchetes son opcionais de de tipo *String*.

Statement	Description
fail([message])	Fai que o método falle. Pode ser útil para comprobar que non se executa unha parte do código.
assertTrue([message,] boolean condition)	Comproba que a condición sexa certa
assertFalse([message,] boolean condition)	Comproba que a condición sexa falsa
assertEquals([message,] expected, actual)	Comproba que expected e actual sexan iguais. Nos Arrays, comproba si son o mesmo array, xa que comproba a referencia

	non os valores.
assertEquals([message,] expected, actual, tolerance)	Comproba que dous *double* ou *float* son iguais, tendo en conta *tolerance* decimais
assertNull([message,] object)	Comproba que o obxecto é null
assertNotNull([message,] object)	Comproba que o obxecto non é null
assertSame([message,] expected, actual)	Comproba que *expected* e *actual* referencian ao mesmo obxecto
assertNotSame([message,] expected, actual)	Comproba que *expected* e *actual* NON referenacian ao mesmo obxecto

Test Suites JUnit

Si dispoñemos de varias clases de Test, podemos combinalas nunha "Test Suite". Executando unha "Test Suite", comprobamos todas as clases de test en esa suite na orde especificada. Unha Test Suite pode ter outras test suites.

O seguinte código amosa o uso de unha Test Suite. Contén dúas clases de proba (MyClassTest and MySecondClassTest). Para engadir máis clases de proba bastará con engadilas á instrucción @Suite.SuiteClasses.

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


@RunWith(Suite.class)
@SuiteClasses({
    MyClassTest.class,
    MySecondClassTest.class })


public class AllTests {
}
```

Inabilitando Tests

Si non queremos que se leve a cabo algún test, bastará con engadirlle a annotation ***@Ignore***. Sen embargo, tamén é posible utilizar ***Assume.assumeFalse,*** ou ***Assume.assumeTrue*** dentro do test para que sexa ignorado. Por exemplo a seguinte sentencia:

Assume.assumeFalse(System.getProperty("os.name").contains("Linux"));

deshabilita o test (o marca como test non válido) si o sistema é Linux.

Tests con Parámetros

JUnit permite deseñar tests aos que se lles poden pasar parámetros de xeito que resulte moito máis cómodo a realización de tests con gran cantidade de datos.

Para poder facer esto, será necesario marcar a clase de test mediante a *annotation* ***@RunWith(Parameterized.class),*** e crear un método estático coa *annotation @Parameters*, que debe suministrar unha *Collection* de Arrays. Cada elemento de esa Collection se utilizará como datos para o test.

Para obter os valores de cada test, podemos utilizar a *annotation* **@Parameter(n)** en atributos públicos.

Vexamos algún exemplo. O seguinte test, comproba o funcionamento dunha clase Multiply, que dispón dun método multiply que multiplica dous enteiros:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {
    // fields used together with @Parameter must be public
    @Parameter(0)
    public int m1;
    @Parameter(1)
    public int m2;
    @Parameter(2)
    public int result;

    // creates the test data
    // Unha Collection é un conxunto de obxectos. En este caso un conxunto de Arrays de tres elementos enteiros
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.asList(data); // transforma o Array de dúas dimensións a unha lista de Arrays
    }

    @Test
    public void testMultiplyException() {
        Multiply tester = new Multiply();
      // De xeito automático, m1 e m2 van tomando todos os valores da lista
```

```
        assertEquals("Result", result, tester.multiply(m1, m2));
    }
}

 // class to be tested
class Multiply {
        public int multiply(int i, int j) {
            return i *j;
        }
 }
```

Tamén é posible utilizar un constructor na clase test para recoller os valores subministrados polo método anotado con @Parameters, en lugar de utilizar a *annotation* ***@Parameter***. Ese constructor debe ter tantos argumentos como os elementos do Array subministrado:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {
    public int m1;
    public int m2;
    public int result;

    // Constructor para recoller os parámetros do test
   public ParametrizedTestFields(int m1,int m2,int r) {
            this.m1=m1;
            this.m2=m2;
            this.result=r;
   }

    // creates the test data
    // Unha Collection é un conxunto de obxectos. En este caso un conxunto de Arrays de tres elementos enteiros
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.asList(data); // transforma o Array de dúas dimensións a unha lista de Arrays
    }
```

```
   @Test
   public void testMultiplyException() {
      Multiply tester = new Multiply();
     // De xeito automático, m1 e m2 van tomando todos os valores da lista
      assertEquals("Result", result, tester.multiply(m1, m2));
   }
}

 // class to be tested
class Multiply {
      public int multiply(int i, int j) {
         return i *j;
      }
 }
```

Regras JUnit

As regras JUnit permiten configurar o comportamento dos tests mediante unha serie de obxectos que permiten crear regras. Estas regras consisten en atributos anotados mediante ***@Rule***, que almacenarán o obxecto de clase **Rule** (*org.junit.rules*) elexido.

Algunhas dos obxectos Rule subminisrados con JUnit son: TemporaryFolder, ExpectedException, ExternalResource, ErrorCollector, Verifier, TestWatcher, TestName, Timeout....

Aquí podes atopar unha referencia:

https://github.com/junit-team/junit4/wiki/Rules

Un exemplo do seu uso:

```
package de.vogella.junit.first;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class RuleExceptionTesterExample {

 // Creamos o obxecto ExpectedException (o Rule) que nos permitirá verificar as excepcións
 @Rule
 public ExpectedException exception = ExpectedException.none();

 @Test
 public void throwsIllegalArgumentExceptionIfIconIsNull() {
  // O test debe lanzar unha excepción IllegalArgumentException para non fallar
  exception.expect(IllegalArgumentException.class);
  // A mensaxe da excepción debe ser "Negative value not allowed" para non fallar
  exception.expectMessage("Negative value not allowed");
  ClassToBeTested t = new ClassToBeTested();
  t.methodToBeTest(-1);
 }
}
```