Curs 20-21

Unitat 06.02
Proves JUnit

Entorns de desenvolupament

IES Jaume II El Just
Tavernes de la Valldigna

Contingut

Introducció

Hem vist que hi ha un cert tipus de proves anomenades funcionals que serveixen per comprovar el funcionament d'un determinat bloc de codi, en general una funció o un mètode d'una classe. També les podem anomenar **proves unitàries**, perquè només comproven un mòdul sense cap interrelació amb la resta.

Si se fan proves unitàries a cada mòdul, després la prova d'integració és més senzilla perquè ja sabem que cada mòdul per separat està funcionant correctament, així que només cal veure que la interacció entre tots els mòduls és també correcta.

Com hem dit, quan parlem de mòdul ens podem estar referint a:

- Una funció independent
- Un mètode d'una classe

En general estem parlant de blocs de codi on hi ha unes entrades i un comportament esperat. També podríem comprovar tota una classe, però sempre ho faríem treballant amb cadascun dels mètodes.

Requeriments de les proves unitàries

Per a resultar efectives, les proves unitàries han de ser:

- ***Automàtiques***: és a dir, amb la menor intervenció manual possible
- ***Completes***: s'ha de revisar tot el codi excepte algun mòdul de funcionament molt simple
- ***Repetibles / Reutilitzables***: s'han de poder executar vàries vegades, i a ser possible han de poder aprofitar-se per provar altres mòduls
- ***Independents***: l'execució d'una prova no ha d'afectar a la resta

Beneficis de les proves unitàries

Si se realitzen amb eficiència i de manera completa, les proves unitàries tenen varis beneficis:

- ***Faciliten les modificacions***: si hi ha canvis en un mòdul tenim un sistema per comprovar si funcionen bé
- ***Faciliten la integració***: si cada mòdul funciona correctament la integració serà més senzilla
- ***Faciliten la documentació***: la documentació generada al llarg de les proves s'integra en la documentació final de l'aplicació
- ***Faciliten la localització d'errors***: lògicament, al ser proves unitàries si falla una prova sabem quin mòdul està fallant

Proves unitàries amb JUnit.

JUnit és un plugin que se pot instal·lar en molts IDEs per realitzar les proves unitàries d'una manera automàtica i senzilla.

Hi ha altres eines per fer proves unitàries en Java, però **JUnit** té l'avantatge de ser de codi obert i pràcticament un estàndard, de manera que és molt senzill trobar informació i exemples en Internet.

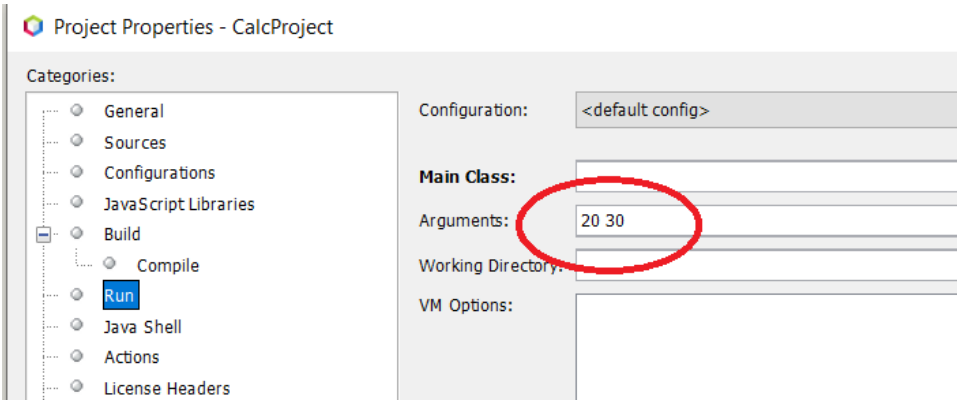
A més **JUnit** permet crear informes en HTML que podem afegir a la documentació de la nostra aplicació, per exemple integrant-la amb els documents generats per **JavaDoc**.

Proves amb NetBeans

Per exemple, anem a fer proves amb una classe ***Calculadora*** que té els mètodes:

- **`suma(op1,op2)`** → torna la suma op1 + op2
- **`resta(op1,op2)`** → torna la resta op1 – op2
- **`multiplica(op1,op2)`** → torna la multiplicació op1 * op2
- **`divideix(op1,op2)`** → torna la divisió op1 / op2
- **`tantpercent(op1,op2)`** → torna el op1% de op2
- **`potencia(op1,op2)`** → torna op1 elevat a op2
- **`getLastResult()`** → torna el resultat de l'última operació que hem fet
- **`getLastOp()`** → torna el nom de l'última operació que hem fet

Per a fer proves unitàries, hauríem de provar cada vegada tots els mètodes amb uns paràmetres d'entrada diferents. Com que li'ls passem com arguments d'entrada a *main*, cada vegada que fem una prova hauríem d'anar a l'opció ***Run*** de les propietats del projecte, canviar els valors dels arguments, i executar l'aplicació.

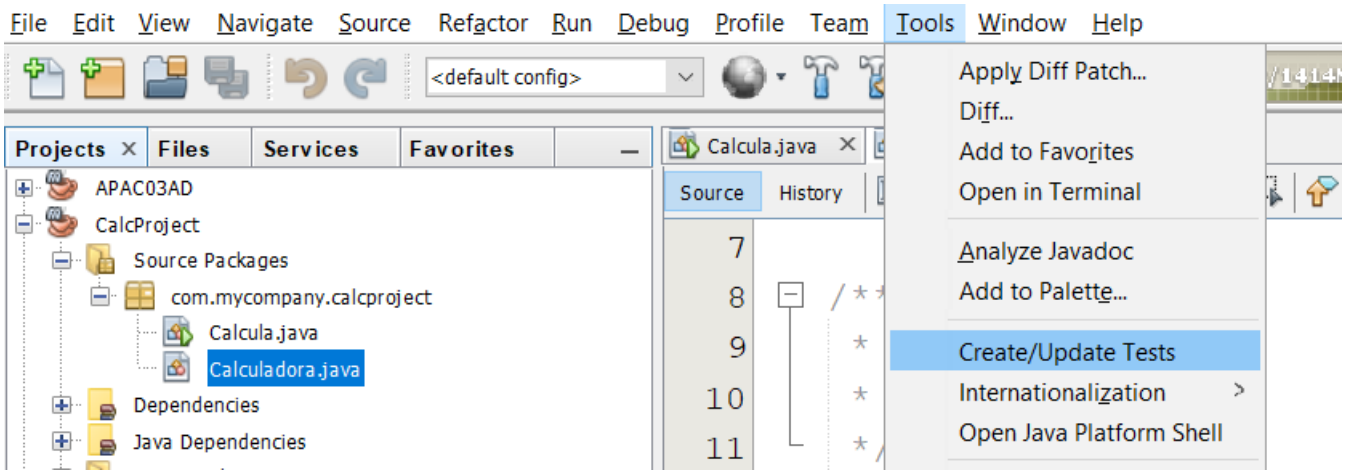


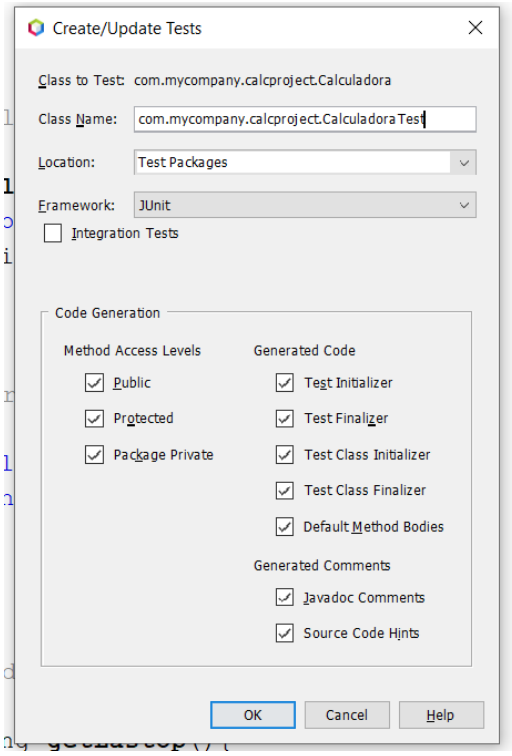
No sembla massa pràctic. Anem a veure una opció millor.

Test JUnit amb NetBeans

JUnit ve integrat amb **NetBeans** com a *test framework* per defecte des de la versió 7.1

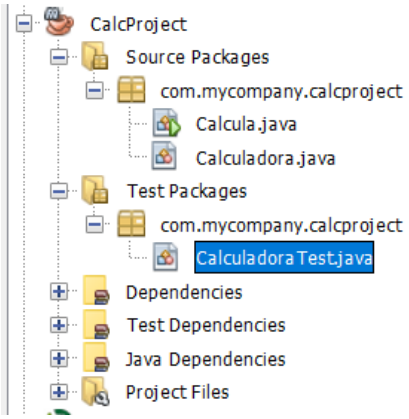
Per provar una classe amb **JUnit**, la seleccionem i anem al menú superior, despleguem l'opció ***Tools*** i seleccionem l'opció ***Create/Update Tests***.





Ens apareix un formulari con podem triar opcions com la classe que volem provar (la que hem seleccionat), en quin lloc volem fer els tests (ens crea un paquet per a fer-los independent de Source Packages), quin framework volem utilitzar (JUnit), i quins mètodes volem comprovar (els públics, els protegits, els privats...) a més d'altres opcions (inicialitzador de test, finalitzador de test...) que deixarem marcades.

Quan polsem OK, veurem que ens genera, com hem dit, un paquet específic per a fer les proves (**Test Packages**) i un arxiu java que se diu igual que la classe que volem provar, però acabada amb *Test*.



Dins de la classe també tenim un mètode per testejar cada mètode de la classe original. També se diu igual però amb *test* davant.

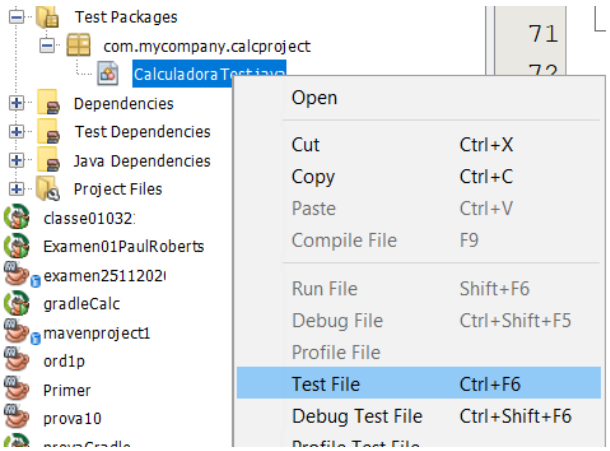
Anem a veure, per exemple, el mètode que ens ha generat per testejar el mètode suma.

```
    /**
     * Test of suma method, of class Calculadora.
     */
    @org.junit.jupiter.api.Test
    public void testSuma() {
        System.out.println("suma");
        float op1 = 0.0F;
        float op2 = 0.0F;
        Calculadora instance = new Calculadora();
        float expResult = 0.0F;
        float result = instance.suma(op1, op2);
        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.");
    }
```

Com podem veure, el mètode consisteix en assignar dos valors a op1 i op2, crear una instància de la classe **`Calculadora`** i cridar al mètode suma passant-li op1 i op2. A més tenim el resultat esperat guardat en una variable **`expResult`**, que després, amb el mètode **`assertEquals`**, compara amb el resultat obtingut. Anem a fer alguns canvis i l'anem a deixar així:

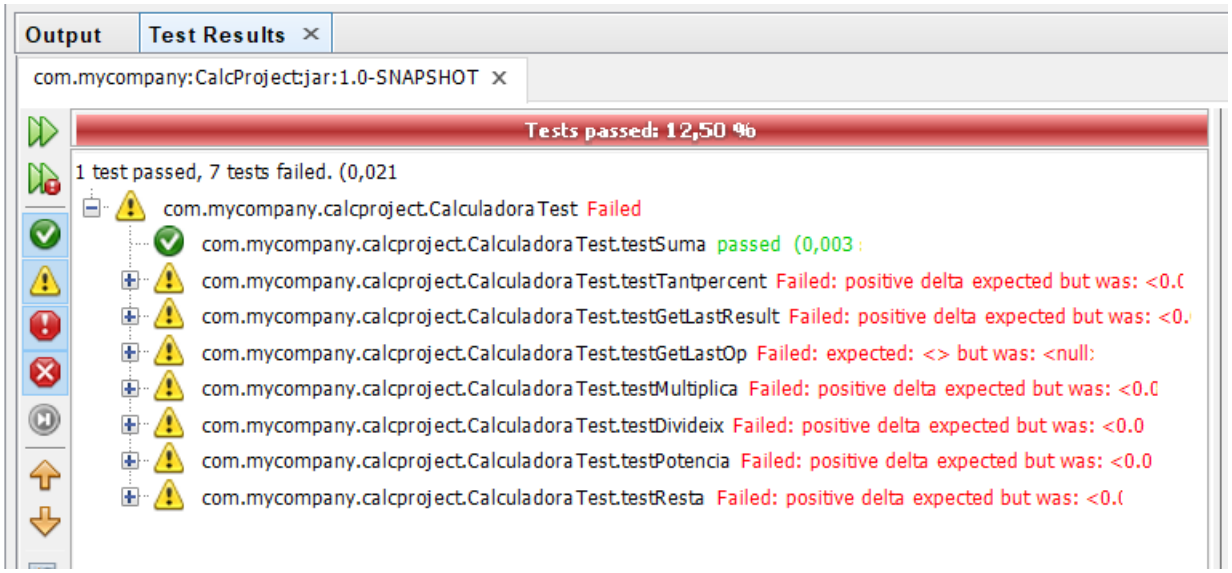
```
public void testSuma() {
        System.out.println("suma");
        float op1 = 10.0F;
        float op2 = 20.0F;
        Calculadora instance = new Calculadora();
        float expResult = 30.0F;
        float result = instance.suma(op1, op2);
        assertEquals(expResult, result);
    }
```

Li hem donat uns valors a op1 i op2, li donem a expResult el valor que esperem que torne la funció suma(), i els comparem amb **`assertEquals`**. Així, si volem, ja podem provar la classe:



- Ens situem sobre la classe de test Calculadora Test.java
- Amb el botó de la dreta obrim el menú contextual
- Seleccionem l'opció **Test File**

Ens mostrarà quins tests han passat, i quins han fallat.



Només ha passat el test el mètode **Suma** perquè no hem corregit la resta de mètodes de prova.

Suma ha passat el test perquè el **`assertEquals`** ha detectat que el valor que torna el mètode original, i el valor esperat que li hem posat manualment, són iguals.

Tenim més opcions per fer comprovacions. Per exemple:

assertTrue() verifica si una expressió booleana és true (verdadera)
assertFalse() verifica si una expressió booleana és false (falsa)
assertNull() verifica si la referencia a un objecte es null (no existeix)
assertNotNull() verifica si la referencia a un objecte no és null (existeix)
assertSame() compara dos referències a objectes i comprova si són iguals
assertNotSame() compara dos referències a objectes i comprova si són diferents
assertEquals() compara dos valors i comprova si són iguals
fails() provoca el fallo de la prova (si hi ha un error o una excepció)

Lògicament, no anem a executar el test cada vegada que volem provar un parell de valors. Dins del mètode del test podem executar tantes operacions **`assertEquals`** com necessitem.

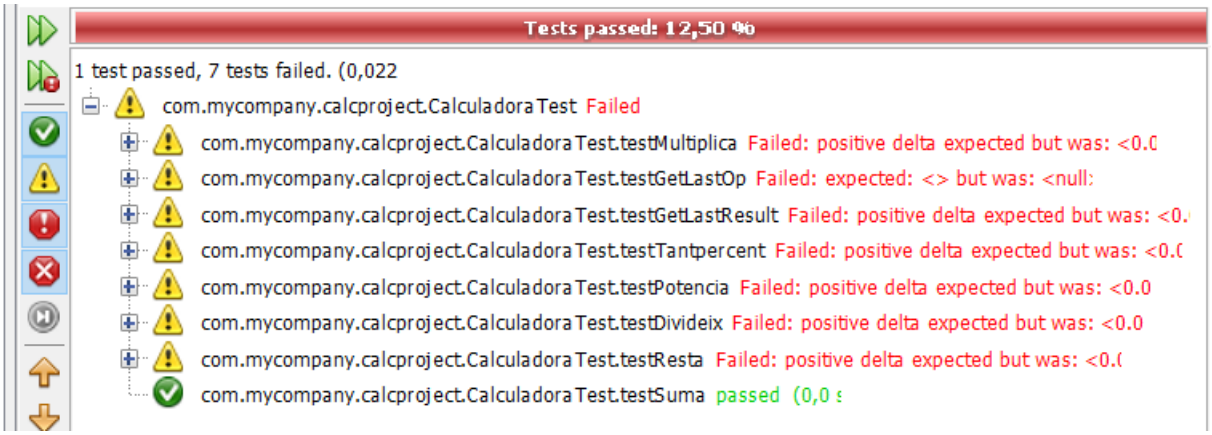
Anem a dissenyar un joc de proves més complet per al mètode suma. Podem començar per les classes d'equivalència. Quines serien? Per exemple, podem provar a...

- Sumar dos números positius
- Sumar un número positiu i un negatiu
- Sumar dos números negatius
- Que un dels dos siga 0, i que els dos siguen 0
- Que un dels tinga decimals
- Que els dos tinguen decimals

No cal que canviem cada vegada els valors i executem el test. Podem deixar el mètode de test com podeu veure a continuació:

```
public void testSuma() {
        Calculadora instance = new Calculadora();
        assertEquals(10+12, instance.suma(10,12));
        assertEquals(8+0, instance.suma(8,0));
        assertEquals(0+0, instance.suma(0,0));
        assertEquals(14+(-6), instance.suma(14,-6));
        assertEquals(-8+5), instance.suma(-8,5));
        assertEquals(-4+(-7), instance.suma(-4,-7));
        assertEquals(10.5F+12.10F, instance.suma(10.5F,12.10F));
    }
```

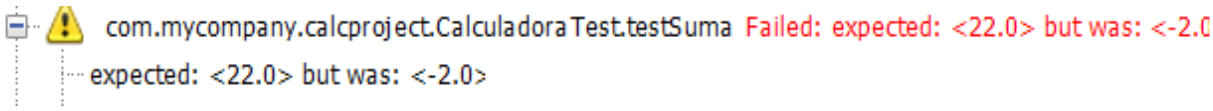
Quan executem el test comprovarà totes les operacions que hem indicat, comparant-les amb el resultat esperat (l'haurem de posar bé, clar) i si passen totes les operacions ens marcarà el test com a superat.



Lògicament, en este cas és un mètode molt simple i no pot fallar. Però sempre, fins i tot en els casos més senzills, ens podem equivocar. Imagineu-vos que copiem i peguem del mètode resta per a fer el mètode suma i ens oblidem canviar el signe de l'operació, per exemple. És a dir, el mètode suma original, sense voler, l'hem deixat així:

```
  public float suma(float op1, float op2){
      float result=op1-op2;  // fixeu-vos que el mètode resta en lloc de sumar
      this.lastResult=result;
      this.lastOp="Suma";
      return result;
    }
```

Estem davant d'un tipus d'error lògic típic, que no se detectarà ni en l'edició ni en la compilació, només fent proves i comprovant que els resultats obtinguts coincideixen amb els esperats. Si ara tornem a fer el test passarà el següent:



S'esperava que la funció suma, si li passem un 10 i un 12, torne 22. Però està tornant -2. El test detecta el fallo, ens avisa, i ens adonem que hem posat mal el signe en el mètode original.

Problemes amb Junit 5 i NetBeans 11

En principi els mètodes anotats amb BeforeAll, BeforeEach, AfterAll i AfterEach s'haurien d'executar en els següents moments:

- BeforeAll → abans de començar tots els tests
- BeforeEach → abans de cada test
- AfterEach → quan acaba cada test
- AfterAll → quan acaben tots els tests

Tot i això, després de fer moltes proves sense que funcionaren eixos mètodes, he trobat que la versió 5 de **JUnit** dóna problemes amb la versió 11 de NetBeans. Si volem fer alguna cosa abans de començar els tests (per exemple, crear l'objecte **`instance`** en lloc de fer-ho en cada mètode) hauríem de treballar amb una versió anterior de JUnit. Bàsicament, el que hauríem de fer és:

- Substituir els imports de **`org.junit.jupiter`** per els corresponents de **`org.junit`**

```
import org.junit.After;
import org.junit.Before;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
```

Segurament vos demanarà que afegiu la dependència corresponent. En el meu cas, que he fet el projecte en Maven, m'ha afegit automàticament la següent dependència:

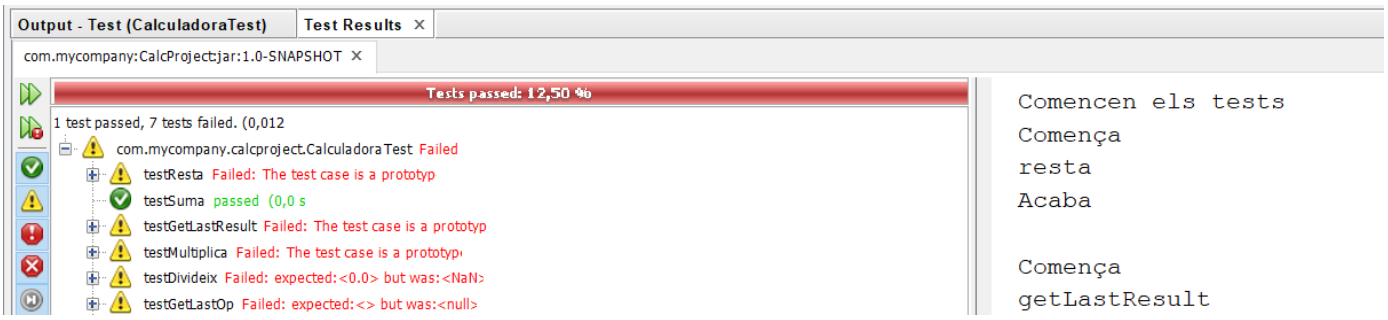
```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
        <type>jar</type>
</dependency>
```

- Substituir les anotacions @BeforeAll per @BeforeClass, @AfterAll per @AfterClass, @BeforeEach per @Before, i @AfterEach per @After

- Afegir una altra vegada el 0.0 final que hi havia en els assertEquals i que havíem llevat al treballar amb els Assertions de Jupiter.

```
assertEquals(10+12, instance.suma(10,12),0.0);
assertEquals(8+0, instance.suma(8,0),0.0);
```

Si ho fem així podem posar instruccions println en cada mètode Before i After i comprovareu com s'executen en el seu moment. Podeu veure en la part dreta de la finestra inferior com els missatges se van mostrant en la pantalla.



Més informació sobre JUnit.

Més informació sobre totes les possibilitats de JUnit en el següent enllaç:

https://netbeans.apache.org/kb/docs/java/junit-intro.html

Sobre JUnit 5

https://junit.org/junit5/docs/current/user-guide/

Vídeos sobre JUnit en Eclipse

https://www.youtube.com/watch?v=EOkoVm3rtNQ&list=PLTd5ehIj0goML37B7s9I9iN2zhJCfxJBC

Videos sobre JUnit en NetBeans

https://www.youtube.com/watch?v=b1SqemeSgSU&ab_channel=JuanCarlosDiaz