Présentation : Test en Java

Mercredi 24 février 2021

Antoine Flotte (aflotte@excilys.com)
Tomy Bezenger (tbezenger@excilys.com)

Excilys
Développeurs de passion

Tests unitaires avec JUnit

JUnit est un framework de tests unitaires pour Java.

Lorsque un projet Maven est créé, il est automatiquement importé dans le pom.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.1</version>
  <scope>test</scope>
</dependency>
```

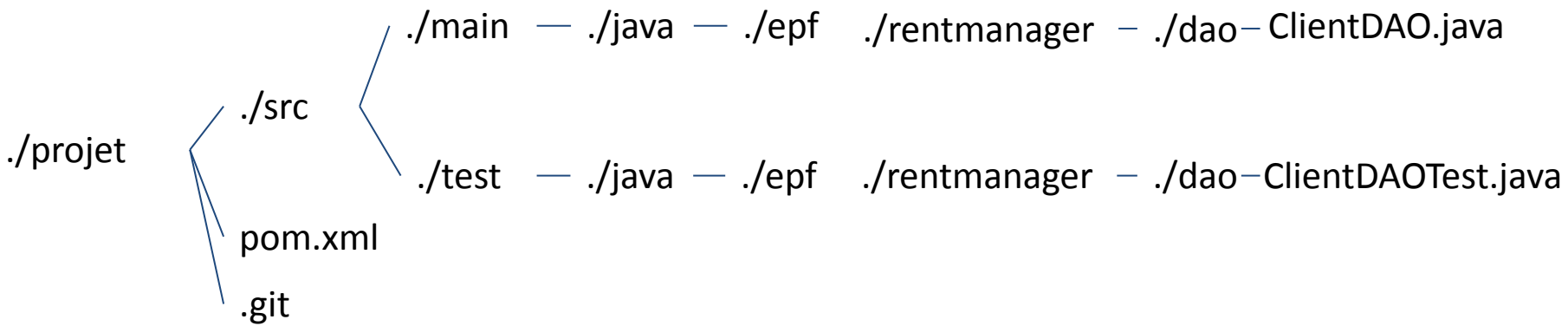
On utilisera ici JUnit 4. Il se peut que votre projet ait importé JUnit 3 (version 3.x.x) par défaut, pensez à changer si c'est le cas.

Le scope définit les limites d'utilisation d'une dépendance au sein du projet. On précise donc que JUnit sera utilisé uniquement dans les tests, mais ce n'est pas une obligation.

Structure dans le projet

Dans le projet, vous avez votre racine avec .git, pom.xml et un dossier src.
Dedans vous avec le dossier main, et le dossier test.
Dedans vous devez avoir les mêmes dossiers, avec la même structure. Puis vos fichiers NomClasse.java deviennent NomClasseTest.java

Ce n'est pas une obligation mais cela permet une lisibilité amélioré.



Structure dans la classe Test

C’est une classe normale, qui démarre donc par :

```
public class BoardTest {
```

Puis on utilise la librairie JUnit :

```
@Test
public void functionTestCaseTest() {
```

C’est forcément une fonction et une classe public, et la fonction ne retourne jamais rien.

Structure dans la fonction

Pour le nom des fonctions, on fait simple :
nomDeLaFonctionTestée + CasDeTest + “Test”
Pour la nomenclature, soit camelCase soit le snake_case, mais toujours en démarrant avec une minuscule. Le choix de nomenclature est gardé dans tout le projet.

Le snake_case a pour avantage de bien différencier les fonctions de tests des fonctions usuelles.

Structure dans la fonction - méthodes

On va principalement utiliser 3 méthodes au sein de nos tests :

- *assertTrue* : Valide un test si le paramètre est vrai.
- *assertFalse* : Valide un test si le paramètre est faux.
- *assertEquals* : Valide un test si les deux paramètres sont équivalents.

Ces méthodes sont statiques, il faut donc les importer de manière statique.

```
@Test
public void returnsOneTest() {
    assertEquals( expected: 1, board.returnsOne());
}

@Test
public void isEvenTest() {
    assertTrue(board.isEven( number: 2));
    assertFalse(board.isEven( number: 3));
}
```

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
```

Structure dans la fonction - *assertEquals*

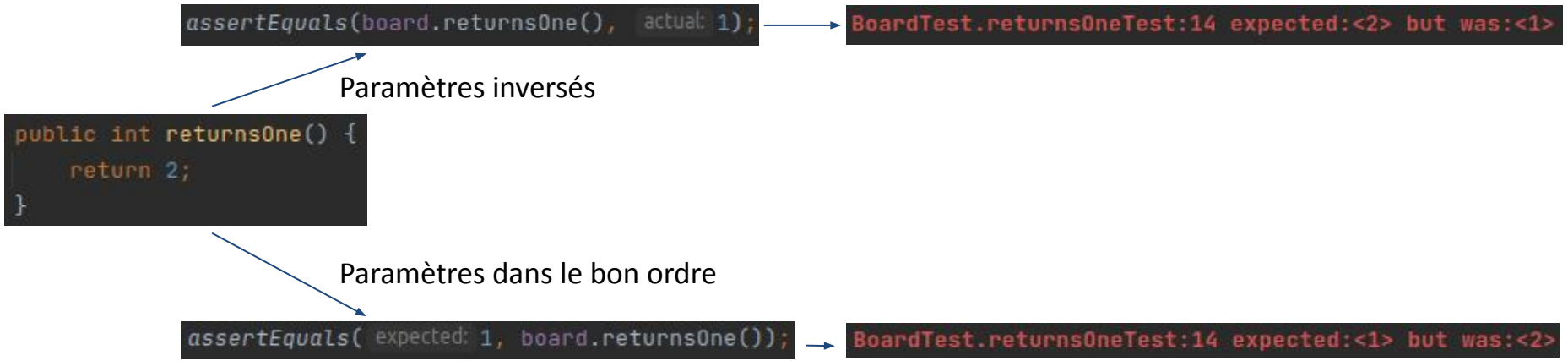
assertEquals entre deux objets utilise les méthodes *equals* des objets, n'oubliez donc pas de les définir.

```java
@Test
public void equalsTest() {
    assertEquals(board, new Board());
}
```

```
[ERROR] Failures:
[ERROR]   BoardTest.equalsTest:25 expected:<ensta.Board@4678c730> but was:<ensta.Board@6767c1fc>
[INFO]
[ERROR] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0
```

Structure dans la fonction - *assertEquals*

Le premier paramètre de *assertEquals* est le résultat attendu, et le second est l'appel à la méthode. D'un point de vue compilation ça ne change rien, mais ça évite des confusions lors de la lecture de la console.



Structure dans la fonction

Mis à part les méthodes type getter/setter/equals, chaque méthode doit être testée.
Chaque cas d'utilisation doit être testé afin de ne rien oublier.

Équivalents

```java
public int returnsOne() {
    return 1;
}

public boolean isEven(int number) {
    return (number % 2 == 0);
}
```

```java
public int returnsOne() {
    return 1;
}

public boolean isEven(int number) {
    if (number % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

```java
@Test
public void returnsOneTest() {
    assertEquals( expected: 1, board.returnsOne());
}

@Test
public void isEvenTest() {
    assertTrue(board.isEven( number: 2));
}

@Test
public void isEvenWhenOddTest() {
    assertFalse(board.isEven( number: 3));
}
```

Excilys
Développeurs de passion

Exceptions levées

Si on veut tester qu'un appel retourne bien une exception il y a plusieurs méthodes :

Faire un try-catch avec un *fail* dans le try :

```
@Test
public void racineCarreeNegativeShouldThrowExceptionTest() {
    try {
        board.racineCarree( number: -10);
        fail("Should have thrown and exception.");
    } catch (IllegalArgumentException e) {
        assertTrue( condition: true);
    }
}
```

Utiliser l'annotation *Test* avec une variable *expected* :

```
@Test(expected = IllegalArgumentException.class)
public void racineCarreeNegativeShouldThrowExceptionTest() {
    board.racineCarree( number: -10);
}
```

Préparation des tests

Il est parfois nécessaire d'effectuer une action avant chaque test, ou bien à la fin de la suite de tests. JUnit permet ceci grâce aux annotations suivantes :

```java
@BeforeClass
public static void beforeAll() {
    System.out.println("Before all tests");
}

@Before
public void beforeEach() {
    System.out.println("Before each test");
}

@After
public void afterEach() {
    System.out.println("After each test");
}

@AfterClass
public static void afterAll() {
    System.out.println("After all tests");
}
```

```
Before all tests
Before each test
Test even
After each test
Before each test
Test odd
After each test
Before each test
Test throw
After each test
Before each test
Test one
After each test
After all tests
```

Les annotations *BeforeAll* et *AfterAll* sont utilisées avec des méthodes statiques.

Le nom de la méthode importe peu, mais il est quand même préférable d'en utiliser un cohérent.

On remarquera que les tests sont appelés dans un ordre plus ou moins aléatoire.