Manuel du développeur

Test unitaire en Java avec JUnit 4

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

1. Introduction

JUnit est un framework open source de développement et d'exécution de tests unitaires pour le langage de programmation Java. JUnit est disponible sur tous les Environnements de Développement Intégré (IDE) et donc utilisable directement pour tout projet Java. JUnit est le framework le plus populaire, mais le framework TestNG est une alternative équivalente. Nous présentons ici un manuel simplifié.

1.1 Exemple introductif de cas de test

Le programme suivant fournit des méthodes statiques pour faire des calculs arithmétiques d'addition et de division.

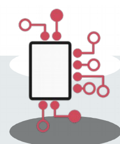
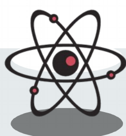
```
1    public final class Calculator {
2       public static int add( int operand1, int operand2 ) {
3          return operand1 + operand2;
4       }
5       public static int div( int operand1 , int operand2 ) {
6          return operand1 / operand2;
7       }
8    }
```

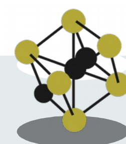
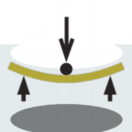
La classe de test suivante code un exemple de cas de test pour les deux opérations `add` et `div` sous la forme de deux méthodes de test unitaire.

```
1    public final class CalculatorTest {
2       @Test
3       public void testAdd() {
4          assertEquals(3, Calculator.add(1, 2));
5       }
6       @Test
7       public void testDiv() {
8          assertEquals(1, Calculator.div(3, 2));
9       }
10   }
```

2. Les éléments de base de JUnit

JUnit définit un certain nombre d'éléments pour simplifier l'écriture de tests, parmi lesquels les annotations et les assertions.

2.1 Les annotations

La création de test avec JUnit utilise les annotations afin de rendre l'écriture du code moins verbeuse. La Table 1 donne un aperçu des annotations de JUnit.

Table 1 : Liste des annotations JUnit.

Annotation	Description
@Test **public void** should_do_when_case()	Marque une méthode comme contenant du code de test.
@Before **public void** setup()	Marque une méthode pour qu'elle s'exécute avant chaque méthode de test de la classe.
@After **public void** tearDown()	Marque une méthode pour qu'elle s'exécute après chaque méthode de test de la classe.
@BeforeClass **public static void** setupBeforeClass()	Marque une méthode statique pour qu'elle s'exécute avant la première méthode de test de la classe. Elle ne s'exécute qu'une seule fois par classe.
@AfterClass **public static void** tearDownAfterClass()	Marque une méthode statique pour qu'elle s'exécute après la dernière méthode de test de la classe. Elle ne s'exécute qu'une seule fois par classe.
@Test(expected=*Exception.class*) **public void** should_do_when_case()	Marque une méthode de test pour vérifier que la méthode testée lève bien l'exception attendue.
@Test(timeout=*100*) **public void** should_do_when_case()	Marque une méthode de test pour vérifier que la méthode testée s'exécute dans le temps imparti.

2.2 Les assertions

Une assertion automatise le verdict d'un test unitaire pour décider si la méthode testée passe avec succès le test. La Table 2 récapitule les principales assertions JUnit. Ce sont des méthodes statiques de la classe `Assert`. Chaque méthode existe en deux versions, avec ou sans un premier paramètre contenant le message qui sera affiché en cas d'erreurs. Par exemple, la première assertion de la Table 2 existe sous la forme `assertTrue(`*`condition`*`)` et `assertTrue(message, `*`condition`*`)`.

Table 2 : Liste des assertions de JUnit.

Fonction	Description
assertTrue(*condition*)	Vérifie que la condition en paramètre est vraie.
assertFalse(*condition*)	Vérifie que la condition en paramètre est fausse.
assertEquals(*expected*, *actual*)	Teste si les valeurs sont égales en utilisant la méthode `equals()`. Note : pour les tableaux, seule la référence est vérifiée, pas le contenu.
assertSame(*expected*, *actual*)	Vérifie que les deux variables pointent vers le même objet en utilisant l'opérateur ==.
assertArrayEquals(*expected*, *actual*)	Teste si les valeurs des éléments sont les mêmes dans les deux tableaux.
assertEquals(*expected*, *actual*, *tolerance*)	Vérifie l'égalité entre doubles ou flottants à une valeur de précision près.
assertNull(*object*)	Vérifie que l'objet en paramètre est bien « null ».
assertNotNull(*object*)	Vérifie que l'objet en paramètre n'est pas « null ».
assertNotSame(*expected*, *actual*)	Vérifie que les deux variables ne pointent pas vers le même objet.

2.3 Création de cas de test

Cette section présente les annotations et les assertions de JUnit à travers un cas

de test de la classe `List` de Java.

```java
1    public final class JUnitTest {
2      private List<String> _list;
3
4      @BeforeClass
5      public static void oneTimeSetUp() {
6        System.out.println("@BeforeClass - oneTimeSetUp");
7      }
8
9      @AfterClass
10     public static void oneTimeTearDown() {
11       System.out.println("@AfterClass - oneTimeTearDown");
12     }
13
14     @Before
15     public void setUp() {
16       _list = new ArrayList<>();
17       System.out.println("@Before - setUp");
18     }
19
20     @After
21     public void tearDown() {
22       _list.clear();
23       System.out.println("@After - tearDown");
24     }
25
26     @Test
27     public void testEmptyList() {
28       assertTrue(_list.isEmpty());
29       System.out.println("@Test - testEmptyList");
30     }
31
32     @Test
33     public void testOneItemList() {
34       _list.add("itemA");
35       assertEquals(1, _list.size());
36       System.out.println("@Test - testOneItemList");
37     }
38   }
```

Le résultat de l'exécution du fichier de test donne la trace suivante :

```
@BeforeClass - oneTimeSetUp
@Before - setUp
@Test - testEmptyList
@After - tearDown
@Before - setUp
@Test - testOneItemList
@After - tearDown
@AfterClass - oneTimeTearDown
```

2.4 Tester le contrat d'une méthode

Le code d'une méthode de test a pour but de vérifier que le contrat d'une méthode métier est respecté pour un cas de test. Par exemple, si la méthode à tester est :

```java
1    public final class JUnit1 {
2      public static int mul( int operand1, int operand2) {
```

```
3          return operand1 * operand2;
4       }
5    }
```

Une méthode de test possible est :

```
1    public final class JUnit1Test {
2       public void should_return_zero_when_multipling_zero_by_zero() {
3          assertEquals(0, JUnit1.mul(0,0));
4       }
5    }
```

Dans la mesure du possible, il faut limiter les méthodes de test à une seule assertion de manière à avoir une bonne identification de l'erreur.

2.5 Tester la durée d'une méthode

Le paramètre `timeout` de l'annotation `@Test` permet de tester la durée maximale de l'exécution d'une méthode en millisecondes. Par exemple, la méthode à tester est :

```
1    public final class JUnit2 {
2       public static void infiniteLoop() {
3          while (true);
4       }
5    }
```

Une méthode de test est :

```
1    public final class JUnit2Test {
2       @Test(timeout = 1000)
3       public void should_throw_timeout_when_infinite_loop() {
4          JUnit2.infiniteLoop();
5       }
6    }
```

Dans le cas précédent, la méthode `infinityLoop()` ne finira jamais, donc le moteur JUnit marquera le test en échec après 1000 millisecondes.

2.6 Tester la levée d'exception

Le paramètre `expected` de l'annotation `@Test` définit le type d'exception qui est attendu si le test se passe bien. Un sous-type dérivé du type de l'exception attendue est aussi accepté.

Dans l'exemple suivant, le test a pour but de vérifier qu'il y a bien une exception de type `ArithmeticException` qui est levée quand il y a une division par 0. La méthode à tester est :

```
1    public final class JUnit3 {
2       public static int div( int o1 , int o2) {
3          return o1 / o2;
4       }
5    }
```

Le code de la méthode de test est :

```
1    public final class JUnit3Test {
```

```
2       @Test(expected = ArithmeticException.class)
3       public void should_throw_exception_when_dividing_by_zero() {
4          JUnit3.div(0,0);
5       }
6    }
```

Dans ce cas, le test terminera sur un succès puisqu'il y a levée d'une exception.

3. Nommer les tests unitaires

Le nom des méthodes de test est plus important qu'il n'y paraît. Quand un test échoue, le développeur veut comprendre ce qu'il vient de casser sans avoir à rentrer dans le code du test. Un nom de test aussi simple que `testAdd` n'est pas assez expressif. Si le test échoue nous savons seulement que quelque chose a cloché dans la méthode `add()`. Mais au-delà de cela, nous ne pouvons rien dire sur la cause sauf à écrire des messages d'erreur très informantif dans les instructions « assert », ce que les développeurs rechignent à faire. Le nom d'une méthode de test doit porter l'intention du test. Le lecteur doit comprendre ce qui est testé tels que le contexte du test et les attendus ou l'action effectuée.

Il existe plusieurs conventions pour nommer les méthodes de test. Nous recommandons celle qui est basée sur le patron should-when :

```
should_<ComportementAttendu>_when_<EtatTesté>
```

Cette convention de nommage renvoie à la composition du corps d'une méthode de test qui distingue trois parties : given-when-then.

```
1    @Test
2    public void should_return_when_doing_this() throws Exception {
3       // Given : Étant donné ce contexte de test
4       // When : Quand j'exécute la fonctionnalité testée
5       // Then : je m'attends à ce résultat.
6    }
```

La première partie « should » décrit l'oracle, c'est-à-dire le résultat attendu du test ou l'action produite. Elle correspond à la description de la partie « then » du corps du test. La seconde partie « when » décrit la fonctionnalité à tester et décrit la partie « when » du corps du test.

Nous recommandons aussi l'utilisation de la notation snake_case pour rendre ce nom très lisible.

Voici quelques exemples de noms construits avec cette convention :

- `should_return_success_when_comparing_two_identical_annotations`
- `should_remove_suffix_when_getting_file_basename`
- `should_throw_exception_when_loading_bad_file_format`
- `should_fail_to_withdrawmoney_when_invalid_account`
- `should_fail_to_valid_when_mandatory_fields_are_missing`

- `should_increase_balance_when_deposit_is_made`

Utiliser les mots « should » et « when » sur chaque test peut paraître répétitif. Mais, le but est de forcer le développeur à se poser ces deux questions à chaque fois qu'il écrit une méthode de test.

Certaines conventions incluent en plus le nom de la méthode testée. Mais, nous ne les recommandons pas. Le développeur sait que lorsque qu'il procède à un changement de nom d'une méthode lors d'un refactoring, il ne fait pas en même temps le changement du nom des méthodes qui la teste. Comme, il n'y a aucun mécanisme qui l'automatise dans les IDE, on se retrouve donc avec des noms de méthode de test qui ne correspondent plus à ceux des méthodes sources.