Test unitaire

Pratique de la programmation orientée-objet

Michel Schinz – 2015-02-16

Unités

Tout programme conséquent est composé de plusieurs **unités**, qui doivent idéalement être aussi indépendantes que possible les unes des autres. Une unité est composée d'un petit nombre (souvent 1) de classes indissociables.

Par exemple, dans un programme de jeu de Monopoly, on peut imaginer avoir une unité représentant la banque, une autre représentant le plateau de jeu, etc.

Test unitaire

Le **test unitaire** ou **test par unité** (*unit testing*) consiste à écrire des petits programmes s'assurant que chaque unité se comporte comme elle le devrait.
Ces tests doivent être automatiques, dans le sens où il ne doit pas être nécessaire qu'un humain examine le résultat de leur exécution pour savoir qu'un problème est survenu. Il est alors possible de les lancer très fréquemment – p.ex. après chaque modification – et de détecter les problèmes dès leur apparition.

JUnit

JUnit (`http://junit.org/`) est une bibliothèque Java facilitant l'écriture de tests unitaires.
Elle fournit des méthodes pour tester que certaines conditions sont satisfaites, et signaler une erreur dans le cas contraire. De plus, elle fournit une infrastructure permettant de lancer tous les tests contenus dans une classe donnée, et signaler les éventuelles erreurs.
Nous l'utiliserons pour illustrer les idées du test unitaire, mais ces idées ne sont pas spécifiques à JUnit ou à Java.

Utilisation de JUnit

Pour tester une unité à l'aide de JUnit, il convient de définir une (ou plusieurs) classe de test contenant un certain nombre de méthodes de test. Une telle méthode se reconnaît à l'annotation `@Test` qu'on lui attache. Exemple :

```
@Test
public void addition() {
  assertEquals(2, 1 + 1);
}
```

La méthode de test doit utiliser une ou plusieurs méthodes d'assertion fournies par JUnit (`assertEquals`, `assertTrue`, etc.), vérifiant qu'une condition est vraie.

Méthode d'assertion

La classe `Assert` de JUnit fournit des méthodes statiques d'assertion, qui vérifient qu'une condition est vraie et signalent une erreur dans le cas contraire. Exemples :

- `assertTrue(boolean b)` vérifie que `b` est vrai,
- `assertNull(Object o)` vérifie que `o` est nul,
- `assertEquals(Object e, Object a)` vérifie que `a` est égal à `e` au moyen de la méthode `equals`,
- `assertEquals(long e, long a)` vérifie que `e` et `a` sont égaux,
- `assertEquals(double e, double a, double d)` vérifie que la différence entre `e` et `a` est inférieure à `d`,
- etc.

Ordre des arguments

Attention : l'ordre des arguments passés à la méthode `assertEquals` a son importance, le premier étant toujours la valeur attendue (*expected value*), le second la valeur obtenue effectivement (*actual value*).

JUnit en tient compte pour produire ses messages d'erreur.

Par exemple :

```
assertEquals("apple", "orange");
```

produit le message suivant :

```
org.junit.ComparisonFailure:
  expected:<apple> but was:<orange>
```

Test d'exceptions

Il est souvent utile de tester qu'une exception est bien levée, ce que les méthodes d'assertion de JUnit ne permettent pas de faire.

Par contre, il est possible d'ajouter un argument à l'annotation `@Test`, spécifiant qu'une exception donnée est attendue. Exemple :

```
@Test(expected = ArithmeticException.class)
public void divByZero() {
   int x = 1 / 0;
}
```

ne pas oublier!

Un tel test échoue si l'exception donnée n'est pas levée lors de son exécution.

Assertions et JUnit

Attention : même si les méthodes de JUnit ont un nom et un but similaire aux assertions Java (énoncé **`assert`**), les deux sont à utiliser dans des contextes très différents :

- les méthodes JUnit doivent être utilisées exclusivement dans des méthodes de test, elles-mêmes placées dans des classes de test ; ces dernières sont utilisées uniquement durant le développement et ne sont jamais distribuées aux utilisateurs,
- les assertions Java peuvent être utilisées n'importe où dans le programme, et sont généralement incluses dans le programme distribué aux utilisateurs — qui ont toujours la possibilité de les désactiver.

Types de test

Lors de la rédaction de tests pour une unité, il y a trois types principaux de tests auxquels il convient de penser :

- les tests de cas d'erreur, qui vérifient que les erreurs qui doivent être signalées le sont bien, p.ex. lorsqu'un argument invalide est fourni,
- les tests de cas aux limites, qui vérifient que l'unité se comporte bien dans les situations délicates, p.ex. qu'une méthode qui accepte un tableau de taille quelconque fonctionne correctement s'il est vide,
- les tests de cas normaux, qui vérifient que l'unité se comporte bien dans les situations « normales ».

Test d'une méthode de tri

Pour illustrer l'écriture de tests unitaires au moyen de JUnit, écrivons un test pour une méthode de tri de tableau entier. Cette méthode a le profil suivant :

```
static void sort(int[] array);
```

Sa documentation spécifie qu'on lui passe en argument un tableau d'entiers et qu'elle trie ce tableau par ordre croissant des éléments.

Le but est d'écrire un test unitaire qui vérifie que tel est bien le cas.

Test d'une méthode de tri

La première méthode de test de notre classe `SortTest` vérifie un cas aux limites : le tri d'un tableau vide.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class SortTest {
  @Test
  public void sortsEmptyArray() {
    int[] a1 = new int[0];
    int[] a2 = new int[0];
    sort(a2);
    assertArrayEquals(a1, a2);
  }
  // Une autre méthode de test suivra…
}
```

Test d'une méthode de tri

La seconde méthode de test vérifie un cas plus général, avec un tableau non vide.

```
boolean isSorted(int[] array) {
  for (int i = 1; i < array.length; ++i)
    if (array[i] < array[i - 1])
      return false;
  return true;
}
@Test
public void sortsNontrivialArray() {
  int[] a = new int[]{ 4,3,6,1,5,6,4,-1 };
  sort(a);
  assertTrue(isSorted(a));
}
```

Exercice

Les deux tests précédents vérifient que :

1. la version triée d'un tableau vide est un tableau vide,
2. après l'appel à la méthode de tri, le tableau est bien trié.

Est-ce suffisant ?

Pouvez-vous écrire une méthode sort qui passe les tests mais qui soit fausse ?

Si oui, quel(s) test(s) devriez-vous encore écrire pour éviter ce problème ?

Limites du test

Comme l'a dit Edsger Dijkstra :

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

c'est-à-dire :

> *Le test peut être utilisé pour démontrer la présence de problèmes, mais jamais pour démontrer leur absence !*

Il est important de toujours garder cela à l'esprit, et de ne surtout pas conclure qu'un programme est correct parce qu'il passe tous les tests que l'on a pensé à écrire.