Test unitaire

Michel Schinz

2017-02-20

1 Introduction

Lors du développement d'un programme, il est important de s'assurer qu'il fonctionne correctement.

Pour de très petits programmes, cela peut se faire à la fin du développement, en testant le comportement du programme complet. Par contre, pour des programmes de taille plus importante — dont le développement peut prendre plusieurs semaines à plusieurs années — cette solution n'est pas réaliste. D'une part, la probabilité que, une fois terminé, un tel programme fonctionne sans jamais avoir été testé est presque nulle ; d'autre part, les erreurs sont difficiles à localiser lorsqu'elles peuvent se trouver n'importe où dans un gros programme.

Dès lors, le test de programmes non triviaux doit se faire petit à petit : dès qu'une partie individuellement utile du programme, appelée **unité** (*unit*), a été écrite, celle-ci peut (et doit) être testée indépendamment. Si des problèmes apparaissent lors du test d'une unité, ceux-ci doivent être corrigés immédiatement avant de continuer. Une fois le test d'une unité terminé, celle-ci peut être supposée correcte (pour l'instant), et le développement du programme peut se poursuivre avec la prochaine unité.

En procédant de la sorte, le programme final, composé d'unités testées individuellement, a beaucoup plus de chances de fonctionner que si la totalité du code avait été écrite sans être testée.

Cette pratique, consistant à tester individuellement les différentes unités d'un programme, est appelée **test unitaire** (*unit testing*).

2 Test unitaire

Comment tester qu'une unité se comporte correctement ? En écrivant un programme de test l'utilisant et vérifiant que son comportement effectif correspond à celui attendu. Pour illustrer cette idée, nous allons écrire ci-dessous un test pour une classe très simple.

2.1 Classe `Arrays`

Admettons que, dans le cadre du développement d'un programme plus important, nous ayons écrit une classe nommée `Arrays` dotée des trois méthodes statiques suivantes :

- `double min(double[] array)`, qui retourne le plus petit élément du tableau `array` ou lève l'exception `NoSuchElementException` si celui-ci est vide,
- `double average(double[] array)`, qui retourne la moyenne des éléments du tableau `array` ou lève l'exception `IllegalArgumentException` si celui-ci est vide,
- `void sort(double[] array)`, qui trie par ordre croissant les éléments du tableau `array`.

Cette classe constitue une unité, dans la mesure où il est tout à fait possible de la tester individuellement. Dans un langage comme Java, une unité est ainsi souvent une classe unique, ou un petit groupe de classes fortement liées entre elles.

Dans le cas de la classe `Arrays`, plusieurs vérifications méritent d'être faites pour chacune des méthodes. Par exemple, on peut imaginer vérifier que `min` :

1. détermine correctement le minimum d'un tableau contenant plusieurs éléments,
2. détermine correctement le minimum d'un tableau contenant un seul élément, et
3. lève l'exception `NoSuchElementException` lorsqu'on lui passe un tableau vide.

Pourquoi vérifier qu'elle fonctionne avec un tableau contenant *un seul* élément ? Car un tel tableau est le plus petit pour lequel le minimum peut être déterminé, et constitue donc ce qu'on appelle souvent un *cas limite*. Il est fréquent que ces cas-là soient gérés de manière incorrecte, et il est donc important de les tester.

Pour effectuer les trois vérifications ci-dessus, développons une classe de test nommée `ArraysTest`.

2.2 Classe `ArraysTest`

Dans un premier temps, organisons la classe `ArraysTest` de la manière suivante :

- à chaque vérification correspond une méthode statique dont le nom évoque la vérification faite,
- chacune de ces méthodes retourne une valeur booléenne indiquant si le test s'est bien déroulé,

- la méthode principale du programme (`main`) appelle chacune de ces méthodes de vérification et affiche `OK` ou `ERROR !` en fonction du résultat.

La classe ci-dessous est organisée de la sorte, et ne contient pour l'instant qu'une seule méthode de vérification, `minWorksOnNonTrivialArray`, qui correspond à la première vérification mentionnée plus haut :

```
public final class ArraysTest {
  private static boolean minWorksOnNonTrivialArray() {
    double[] a = new double[] {
      1, 2, -12, 20, 0, -1003, -1003.2, 12, 12, -1003.2
    };
    double expectedMin = -1003.2;
    double actualMin = Arrays.min(a);
    return expectedMin == actualMin;
  }

  private static void check(boolean result) {
    System.out.println(result ? "OK" : "ERROR!");
  }

  public static void main(String[] args) {
    check(minWorksOnNonTrivialArray());
  }
}
```

Si la méthode `min` calcule correctement le minimum du tableau qu'on lui passe, ce programme affiche `OK`, sinon il affiche `ERROR !`.

On peut compléter cette classe `ArraysTest` en lui ajoutant une seconde méthode de vérification, `minWorksOnTrivialArray`, qui s'assure que la méthode `min` fonctionne lorsqu'on lui passe un tableau contenant un seul élément :

```
public final class ArraysTest {
  // … comme avant

  private static boolean minWorksOnTrivialArray() {
    double[] a = new double[]{ -12.2 };
    double expectedMin = -12.2;
    double actualMin = Arrays.min(a);
    return expectedMin == actualMin;
  }

  public static void main(String[] args) {
    // … comme avant
    check(minWorksOnTrivialArray());
```

```
  }
}
```

Finalement, on peut ajouter une troisième méthode de vérification, `minFailsOnEmptyArray`, qui vérifie que la méthode `min` lève bien l'exception `NoSuchElementException` lorsqu'on lui passe un tableau vide. Cette méthode est relativement lourde à écrire puisqu'elle doit appeler la méthode `min` puis retourner vrai si elle lève la bonne exception et faux dans les autres cas, c-à-d si elle lève une autre exception ou si elle n'en lève aucune.

```
public final class ArraysTest {
  // … comme avant

  private static boolean minFailsOnEmptyArray() {
    try {
      Arrays.min(new double[0]);
    } catch (NoSuchElementException e) {
      return true;
    } catch (Exception e) {
      return false;             // mauvaise exception
    }
    return false;               // aucune exception
  }

  public static void main(String[] args) {
    // … comme avant
    check(minFailsOnEmptyArray());
  }
}
```

Même s'il est tout à fait possible d'écrire des tests unitaires de cette manière, on imagine facilement que beaucoup de code sera commun à plusieurs tests. Par exemple, chaque fois que l'on désire vérifier qu'une méthode lève bien une exception dans une situation donnée, il faut écrire du code similaire à celui de la méthode `minFailsOnEmptyArray` ci-dessus.

Comme toujours en programmation, lorsqu'on constate une répétition de code, il convient d'essayer de l'extraire (on dit aussi « factoriser ») dans une bibliothèque, afin de pouvoir le réutiliser sans le dupliquer.

Dans le cas du test unitaire, il existe déjà de nombreuses bibliothèques pour la plupart des langages de programmation. Dans le monde Java, la plus populaire d'entre elles est probablement JUnit.

3 Test avec JUnit

JUnit est une bibliothèque facilitant l'écriture de tests unitaire en Java. Même si elle n'est pas forcément la meilleure qui soit, elle a l'avantage d'être simple et bien intégrée à Eclipse, raison pour laquelle nous l'utilisons dans ce cours.

3.1 Principes de base

Un test unitaire JUnit prend la forme d'une classe composée d'un certain nombre de méthodes de test. L'organisation générale d'une telle classe est donc similaire à celle de la classe `ArraysTest` écrite plus haut.

Les méthodes de test sont identifiées par l'annotation `@Test` (du paquetage `org.junit`) qui leur est attachée. Il doit s'agir de méthodes d'instance, sans arguments ni valeur de retour — leur type de retour est `void`. Lors de l'exécution d'un test, toutes les méthodes annotées de la sorte sont exécutées par JUnit, dans un ordre quelconque.

> **Attention :** L'ordre d'exécution des tests JUnit étant quelconque, il est très important de ne jamais introduire de dépendance entre deux tests. C'est-à-dire que le comportement d'un test ne doit jamais dépendre de l'exécution préalable d'un autre test.

On l'a vu, la plupart des méthodes de test ont pour but de vérifier que le résultat effectif produit par l'unité en cours de test correspond bien au résultat attendu. Pour ce faire, JUnit offre plusieurs méthodes statiques dans la classe `Assert`, dont le nom commence par `assert`. La plus importante d'entre elles, `assertEquals`, vérifie l'égalité de deux valeurs, et fait échouer le test en cas de différence. Cette méthode est surchargée et il en existe un très grand nombre de variantes, chacune permettant de comparer des valeurs d'un type différent, p.ex. deux entiers de type `int`, deux objets, etc.

> **Attention :** Ne confondez pas les méthodes de JUnit dont le nom *commence* par `assert` (p.ex. `assertEquals`) et les assertions Java, introduites par le mot-clef `assert`. Les méthodes de JUnit ne doivent être utilisées que dans les classes de test, jamais dans le code principal, tandis que les assertions Java ne sont généralement pas utilisées dans les classes de test, mais uniquement dans le code principal.

3.2 Adaptation de la classe `ArraysTest`

La classe `ArraysTest` peut être adaptée pour utiliser JUnit de la manière suivante :

- les méthode `main` et `check`, désormais inutiles, sont supprimées,
- les méthodes de test sont annotées avec l'annotation `@Test`, rendues publiques et non statiques,

- le type de retour des méthodes de test est changé en `void` et la comparaison entre la valeur attendue et la valeur obtenue est faite au moyen de la méthode `assertEquals` de JUnit.

Par exemple, la première méthode est transformée ainsi :

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

import java.util.NoSuchElementException;
import org.junit.Test;

public class ArraysTest {
  @Test
  public void minWorksOnNonTrivialArray() {
    double[] a = new double[] {
      1, 2, -12, 20, 40, -1003, -1003.2, 12, 12, -1003.2
    };
    assertEquals(-1003.2, Arrays.min(a), 0);
  }

  // … autres tests à venir
}
```

La variante de la méthode `assertEquals` utilisée ici permet de vérifier que deux nombres en virgule flottante (de type `double`) ont la même valeur, à une tolérance près. Les trois arguments passés à cette méthode sont, dans l'ordre :

1. le nombre attendu, ici -1003.2,
2. le nombre effectivement obtenu, ici celui retourné par la méthode `min`,
3. la différence tolérée entre le nombre attendu et obtenu, ici 0.

La méthode `assertEquals` compare le nombre attendu et le nombre obtenu, et si leur différence en valeur absolue est plus grande que la tolérance, provoque l'échec du test.

> **Attention :** Lors de l'utilisation de la méthode `assertEquals`, il faut prendre garde à l'ordre des arguments : le premier est la valeur attendue, le second la valeur effectivement obtenue. Cet ordre est important car il influence les messages d'erreurs.

Comme nous le verrons plus loin, la tolérance est importante en raison des erreurs d'arrondis qui peuvent se produire lors des calculs avec des nombres à virgule flottante. Ici, étant donné que la méthode `min` ne fait aucun calcul, on s'attend à ce qu'elle retourne *exactement* la valeur minimale du tableau, raison pour laquelle la tolérance est fixée à 0.

La seconde méthode de test peut être adaptée aussi facilement que la première :

```
public class ArraysTest {
  // … comme avant
  @Test
  public void minWorksOnTrivialArray() {
    double[] a = new double[]{ -12.2 };
    assertEquals(-12.2, Arrays.min(a), 0);
  }
}
```

La troisième méthode de test, qui vérifie que l'exception `NoSuchElementException` est bien levée lorsque la méthode `min` est appelée avec un tableau vide, peut être considérablement simplifiée grâce à JUnit. En effet, l'annotation `@Test` attachée aux méthodes accepte un argument nommé `expected` et donnant l'exception attendue :

```
public class ArraysTest {
  // … comme avant
  @Test(expected = NoSuchElementException.class)
  public void minFailsOnEmptyArray() {
    Arrays.min(new double[0]);
  }
}
```

Un test annoté de la sorte est considéré comme réussi si et seulement si son exécution provoque la levée de l'exception attendue, ici `NoSuchElementException`. Si la méthode termine sans lever d'exception ou lève une autre exception que celle attendue, le test échoue.

Notez que le nom de l'exception passé en argument à l'annotation `@Test` est suivi de `.class`, pour des raisons qu'il n'est pas important de comprendre à ce stade.

3.3 Test de la méthode `average`

La méthode `average` de la classe `Arrays` peut être testée de manière similaire à la méthode `min`, en écrivant trois méthodes de test :

1. la première vérifiant que la méthode `average` calcule correctement la moyenne d'un tableau contenant plusieurs éléments,
2. la seconde vérifiant que la méthode `average` calcule correctement la moyenne d'un tableau ne contenant qu'un seul élément (cas limite),
3. la dernière vérifiant que la méthode `average` lève bien l'exception `IllegalArgumentExcept` lorsqu'elle reçoit un tableau vide.

Par exemple, la première de ces méthodes peut s'écrire ainsi :

```
public class ArraysTest {
  // … comme avant
  @Test
  public void averageWorksOnNonTrivialArray() {
    double[] a = new double[] { 1000, 0.2, 0 };
    assertEquals(333.4, Arrays.average(a), 0);
  }
}
```

En exécutant ce test, on constate toutefois qu'il échoue ! Cela est surprenant, dans la mesure où la moyenne de 1000, 0.2 et 0 est bien 333.4.

Malheureusement, en raison des erreurs d'arrondi, la valeur calculée par Java au moyen du type `double` est 333.40000000000003. Dès lors, il convient ici de passer une autre valeur que 0 comme tolérance à la méthode `assertEquals`, par exemple 10^{-10} :

```
assertEquals(333.4, Arrays.average(a), 1e-10);
```

Les deux autres méthodes de test pour la méthode `average` s'écrivent sans difficulté et sont laissées en exercice.

3.4 Test de la méthode `sort`

Pour terminer la classe `ArraysTest`, il convient encore de tester la méthode `sort`. Etant donné qu'elle ne peut pas lever d'exception, on peut se contenter de tester les cas suivants :

1. le cas limite, à savoir le tri d'un tableau vide,
2. au moins un cas « normal », à savoir le tri d'un tableau non vide et initialement non trié.

Le test du cas limite est trivial, puisqu'il consiste simplement à appeler la méthode `sort` avec un tableau vide. Si aucune exception n'est levée, le test est réussi :

```
public class ArraysTest {
  // … comme avant
  @Test
  public void sortWorksOnTrivialArray() {
    Arrays.sort(new double[0]);
  }
}
```

Pour le test du cas normal, on peut imaginer procéder ainsi :

1. on ajoute à la classe `ArraysTest` une méthode `isSorted` vérifiant que le tableau qu'on lui passe est bien trié,

2. on écrit un test qui échoue si cette méthode ne retourne pas vrai lorsqu'on l'applique au tableau *après* l'appel à la méthode `sort`, ce qui peut se faire au moyen de la méthode `assertTrue` de JUnit.

On obtient le résultat suivant :

```
public class ArraysTest {
  // … comme avant
  private boolean isSorted(double[] array) {
    for (int i = 1; i < array.length; ++i) {
      if (! (array[i - 1] <= array[i]))
        return false;
    }
    return true;
  }

  @Test
  public void sortWorksOnNonTrivialArray() {
    double[] a = new double[] {
      3, -5, 6, 2, 77.2, 2, 17, -5, -1500
    };
    Arrays.sort(a);
    assertTrue(isSorted(a));
  }
}
```

Admettons qu'en exécutant ce test, on constate que JUnit ne signale aucune erreur. Peut-on en conclure que la méthode `sort` est correcte ?

Malheureusement non ! Par exemple, la méthode `sort` ci-dessous est clairement incorrecte, puisqu'elle ne trie pas le tableau qu'elle reçoit mais remplace chacun de ses éléments par 0. Pourtant, étant donné qu'après son exécution le tableau est effectivement (trivialement) trié, elle passe le test `sortWorksOnNonTrivialArray` avec succès…

```
public static void sort(double[] array) {
  // "Tri par effacement" :
  // tous les éléments sont mis à 0!
  for (int i = 0; i < array.length; ++i)
    array[i] = 0;
}
```

Comme cet exemple l'illustre, il faut être très prudent lors de l'écriture de tests unitaires, afin de bien penser à couvrir autant de cas que possible.

De manière générale, il ne faut jamais conclure que si les tests liés à une unité s'exécutent sans erreur, alors cette unité est correcte. Cela n'est vrai que si les tests sont corrects

et exhaustifs, et en pratique il n'est presque jamais possible d'écrire des tests exhaustifs pour des unités réalistes. Cette limitation constitue la principale faiblesse des tests.

L'informaticien néerlandais Edsger Dijkstra a résumé cet état de fait dans une phrase devenue célèbre :

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

Autrement dit, si un test échoue, on peut être certain que l'entité testée et/ou le test lui-même contiennent une erreur. Par contre, si un test passe, on ne peut être certain que l'unité testée soit exempte de problèmes. N'oubliez jamais cela et n'accordez pas une confiance aveugle aux tests !

4 Références

- *Pragmatic Unit Testing in Java 8 with JUnit* de Dave Thomas et coauteurs,
- le site Web de JUnit, en particulier :
  - la documentation de l'API, en particulier :
    - la classe `Assert`, qui définit `assertEquals` et les autres méthodes d'assertion,
    - l'annotation `Test`, qui identifie les méthodes de test.
  - le Wiki, qui donne de nombreux exemples d'utilisation.