POGL 2020 – Test unitaire

1 Test

Effectuer un test c'est : faire tourner un morceau de programme ou un système entier, et contrôler que les résultats obtenus sont bien ceux attendus.

Le test est le moyen le plus simple à mettre en œuvre pour s'assurer qu'un programme fonctionne à peu près comme prévu. C'est un effort minimal et indispensable, à utiliser en toutes circonstances, de l'exercice de TP quelconque au grand projet industriel. Dans certains cas, le test sert même de guide au développement.

Différents types de test ont lieu à différents stades du développement. On a notamment :

— Lors du déploiement d'un système entier, des utilisateurs le testent en conditions réelles (**tests alpha, beta**...). Dans le diagramme en V, cette phase de test apparaît tout en faut à droite et valide le cahier des charges.

— Conjointement à l'écriture du code d'une classe, le **test unitaire** s'intéresse à chaque méthode indépendamment l'une de l'autre. Il s'agit de vérifier, dans des conditions contrôlées, que le comportement de chaque méthode est cohérent avec sa spécification, qui a été déterminée lors de la phase de conception.

— Entre les deux, les **tests d'intégration** testent des scénarios faisant intervenir plusieurs composants du système.

— En phase de maintenance, chaque modification du code appelle des **tests de régression**, qui visent à vérifier que les corrections ou évolutions n'ont pas introduit de problèmes dans les parties qui fonctionnaient précédemment.

Les tests unitaires et tests de régression sont en général nombreux et effectués souvent : ils sont donc automatisés. Dans ce cours, on regardera l'automatisation des tests unitaires pour Java avec l'outil JUnit.

2 Test unitaire

Un test unitaire s'intéresse au comportement d'une unique méthode. Une méthode `m` étant donnée, un test unitaire est défini par :

1. des entrées concrètes, c'est-à-dire un état de l'objet appelant et des paramètres pour l'appel de méthode, ces entrées devant respecter les préconditions de la méthode ;
2. le résultat devant être renvoyé d'après la spécification, si la méthode n'a pas un type de retour `void` ;
3. l'état attendu d'après la spécification, si la méthode modifie des attributs.

Le test consiste à effectuer l'appel de méthode pour les objets donnés en entrée, et de comparer le résultat obtenu au résultat attendu.

— Si les résultats correspondent, alors le comportement du programme est cohérent avec ce qui était attendu. Cela ne signifie pas que le programme est correct à coup sûr, mais de nombreux tests (bien choisis) réussis apportent une certaine confiance.

— Si les résultats ne correspondent pas, alors on vient de mettre au jour une erreur, et le programme doit être corrigé.

Exemple 1. Quatre tests pour la méthode `sqrt` sont donnés par le tableau suivant :

	Entrée	Sortie attendue
Test 1	`4.0`	`2.0`
Test 2	`1.0`	`1.0`
Test 3	`0.0`	`0.0`
Test 4	`1048576.0`	`1024.0`

Il serait absurde de donner un test avec une entrée négative : la précondition de la méthode `sqrt` demande une entrée positive ou nulle, et la spécification ne dit donc rien de ce qui devrait se passer dans le cas contraire. Autrement dit, nous n'avons aucune idée de la sortie qui serait attendue (à supposer qu'il y ait bien une sortie).

Exemple 2. Trois tests pour l'appel de méthode `l.add(e)` sont donnés par le tableau suivant :

	`l`	`e`	`l` attendu après appel
Test 1	vide	1	`{1}`
Test 2	`{1, 2, 4}`	8	`{1, 2, 4, 8}`
Test 3	`{1, 2, 4}`	2	`{1, 2, 4, 2}`

3 JUnit

Avec JUnit, les tests sont définis dans des fichiers `.java` à côté des fichiers de code. Un fichier de test contient une classe publique, et les tests sont des méthodes de cette classe.

Un test est défini par une méthode sans paramètre et sans valeur de retour (sa signature est donc de la forme `public void m()`), et précédée de l'annotation `@Test`.

Une méthode de test comporte trois parties :

1. une initialisation, qui doit construire les entrées à fournir à l'appel de méthode : il faut définir les objets sur lesquels portera le test et les placer dans l'état spécifié ;
2. l'appel de méthode ;
3. un diagnostic, qui doit vérifier le résultat obtenu et les éventuelles modifications ou non-modifications d'objets, en utilisant des assertions parmi les suivantes :

Méthode	**Rôle**
`assertEquals(Object a, Object b)`	Vérifie que les objets a et b sont égaux
`assertSame(Object a, Object b)`	Vérifie que a et b sont des références vers le même objet
`assertNotSame(Object a, Object b)`	Vérifie que a et b ne sont pas des références vers le même objet
`assertNull(Object o)`	Vérifie que l'objet o est `null`
`assertNotNull(Object o)`	Vérifie que l'objet o n'est pas `null`
`assertTrue(boolean e)`	Vérifie que l'expression e est vraie
`assertFalse(boolean e)`	Vérifie que l'expression e est fausse
`fail()`	Provoque l'échec du test

Lorsque tous les tests d'une classe de test ont une partie de leur initialisation en commun, on peut inclure dans la classe une méthode précédée de l'annotation `@Before` qui contient ce code d'initialisation commun.

Lorsque les tests sont effectués, toutes les méthodes annotées par `@Test` sont appelées l'une après l'autre, et l'éventuelle méthode annotée par `@Before` est appelée avant chaque appel d'une méthode de test. Si tous les tests ont réussi, l'outil rapporte un succès. Si des tests échouent (car le résultat n'est pas celui attendu ou car une erreur a eu lieu), l'outil indique quels tests ont posé problème.

Dans les cas simples, la méthode `@Before` redéfinit les principaux objets utilisés avant chaque test, et on peut donc considérer qu'on repart de zéro. Si certaines choses ne sont pas réinitialisées de cette manière, on peut ajouter une méthode annotée par `@After` qui sera appelée après chaque appel d'une méthode de test.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class EnsembleTest {

  // Vérifier que l'ensemble vide a pour cardinal 0.
  @Test
  public void ensembleVideEstVide() {
    // Trois étapes
    // 1. Construire l'exemple (l'ensemble vide)
    Ensemble e = new Ensemble();
    // 2. Appeler la méthode (exécuter le test)
    int res = e.cardinal();
    // 3. Comparer résultat attendu et résultat obtenu
    assertEquals(0, res);
    assertFalse(e.contient(1));
  }

  // Un ensemble à un élément a pour cardinal 1.
  @Test
  public void cardinalSingleton() {
    // 1. On construit un ensemble contenant uniquement l'élément 1
    Ensemble e = new Ensemble();
    e.ajoute(1);
    // 2.
    int res = e.cardinal();
    // 3.
    assertEquals(1, res);
  }

  // Ajouter un élément déjà présent ne change rien
  @Test
  public void ajoutEltDejaPresent() {
    // 1. On construit un ensemble contenant uniquement l'élément 1
    Ensemble e = new Ensemble();
    e.ajoute(1);
    // 2. On ajoute l'élément 1
    e.ajoute(1);
    // 3. On vérifie qu'il n'y a toujours qu'un élément
    assertEquals(1, e.cardinal());
    assertTrue(e.contient(1));
  }
}
```

FIGURE 1 – Un exemple de fichier de test.