Environnements de développement (intégrés)

Tests unitaires, outils de couverture de code

Patrick Labatut
labatut@di.ens.fr
http://www.di.ens.fr/~labatut/

Département d'informatique
École normale supérieure

Centre d'enseignement et de recherche en technologies de l'information et systèmes
École des ponts

Transparents inspirés de ceux du cours de Mihaela Sighireanu en 2006-2007.

Plan

Introduction

Définition

Un test unitaire est une procédure permettant de s'assurer du bon fonctionnement d'une portion (une « unité ») de programme.

En Java, unité = méthode.

Dans certains modèles de cycle de développement logiciels[1], les tests unitaires sont écrits avant le code à tester pour indiquer/spécifier comment ce code va être utilisé.

[1] *Test-driven development*

xUnit

Il s'agit de l'architecture[2] de système de tests unitaires la plus répandue, le nom est en fait dérivé de JUnit (la version pour Java)[3].
Les même principes pour écrire et lancer des tests unitaires s'appliquent quel que soit le langage :

- JUnit pour Java[4]
- CUnit pour le C
- CppUnit pour le C++
- PyUnit pour le Python
- JsUnit pour le JavaScript
- FlexUnit pour Adobe Flex / ActionScript (Flash...)
- PHPUnit pour PHP
- ...

[2] Biblothèques pour l'écriture/l'exécution de tests unitaires
[3] Pour l'historique exact : `http://www.martinfowler.com/bliki/Xunit.html`
[4] TestNG, aussi...

Vocabulaire de xUnit

Assertion : fonction ou macro vérifiant une condition à l'exécution, si la vérification échoue une exception est levée/arrêt du test courant.

Test Fixture : initialisation/terminaison commune à tout les tests unitaires.

Test Suite : une collection de tests unitaires.

Test Execution : ...

Écriture des tests 1/3

En général (en Java) :

- une classe de tests unitaires par classe testée dans le même package,
- dans cette classe, plusieurs méthodes correspondant chacune à un test unitaire d'une méthode de la classe à tester,
- chaque test unitaire utilise une ou plusieurs assertion(s) pour vérifier l'exécution correcte de la méthode correspondante de la classe à tester (sous différentes conditions d'exécution, etc. . .).
- la classe de test contient éventuellement une méthode d'initialisation et une méthode de terminaison utilisée (implicitement) par chacune des méthodes de test.

Écriture des tests 2/3

En pratique :

La classe de test hérite de `junit.framework.TestCase`.

Elle implémente une ou plusieurs méthodes `public void testNomDuTest()` (les tests unitaires).

Elle implémente (éventuellement) une méthode `void setUp()` (au moins `protected`) qui sera invoquée avant chaque test (*Test Fixture*).

Elle implémente (éventuellement) une méthode `void tearDown()` (au moins `protected`) qui sera invoquée après chaque test (*Test Fixture*).

Écriture des tests 3/3

Les assertions suivantes sont disponibles :

- `fail()`
- `assertTrue(condition)` / `assertFalse(condition)`
- `assertEquals(attendu, effectif)` pour type `boolean`/`byte`/`char`/`int`/`long`/`short`/`Object`[5]
- `assertEquals(attendu, effectif, delta)`[6] pour type `float`/`double`[7]
- `assertNotNull(objet)` / `assertNull(objet)`
- `assertNotSame(attendu, effectif)`[8]

+ des variantes pour indiquer un message d'erreur (1er paramètre).

---

[5] utilise `equals()` pour `Object`

[6] Vérifie que `Math.abs(expected - actuel) <= delta`

[7] *Rappel :* ne jamais tester l'égalité de deux nombres flottants résultants d'opérations différentes avec l'opérateur ==.

[8] égalité des références

Exécution des tests 1/2

Pour l'instant, il n'y a aucun point d'entrée dans la classe de tests.

Rajouter :

- `import junit.framework.Test;` et
- `import junit.framework.TestSuite;`

en plus de `import junit.framework.TestSuite`.

Implémenter une méthode `public static Test suite()` avec pour seule instruction :

```
return new TestSuite(NomDeLaClasseDeTests.class);
```

C'est le point d'entrée de la suite de tests.

Exécution des tests 2/2

Implémenter une méthode
`public static void main(String[] args)` avec pour seule instruction[9] :

```
junit.textui.TestRunner.run(suite());
```

Pour lancer le test unitaire, exécuter la classe de test sans oublier de placer la biblothèque JUnit dans le `CLASSPATH`.

[9] Il existe d'autres interfaces (graphiques, en particulier) d'exécution de suite de tests...

Exemple (JUnit 3.x)

```
1  import java.util.ArrayList;
2  import java.util.Collection;
3
4  import junit.framework.Test;
5  import junit.framework.TestCase;
6  import junit.framework.TestSuite;
7
8  public class ArrayListTest extends junit.framework.TestCase {
9      private Collection collection;
10
11     protected void setUp() {
12         collection = new ArrayList();
13     }
14
15     public void testEmptyCollection() {
16         assertTrue(collection.isEmpty());
17     }
18
19     public static Test suite() {
20         return new TestSuite(ArrayListTest.class);
21     }
22
23     public static void main(String[] args) {
24         junit.textui.TestRunner.run(suite());
25     }
26 }
```

Tests unitaires dans Eclipse

Vous pouvez oublier tout ce qui concerne les méthodes `suite()` et `main()`.

[File / New / JUnit Test Case] (Choisir *JUnit 3.8.1 test*)

Eclipse crée un patron de classe de test avec les imports adéquats et éventuellement des squelettes pour `setUp()`/`tearDown()`

Exécution : *[Mouse-R / Run As / JUnit Test]*, une nouvelle vue apparaît indiquant le succès ou l'échec de la suite de tests et, pour chacun des tests unitaires, la cause (éventuelle) d'échec (assertion non vérifée, exception non rattrapée).

Attention : ne pas oublier d'ajouter la bibliothèque JUnit 3.8.1 au *Build Path* du projet.

Démonstration

Problèmes courants 1/2

- Faut-il tout tester ?

  Il est inutile de tester méthodes triviales (par ex. : accesseur/affecteur trivial).

- Comment tester une méthode sans valeur de retour ?

  Tester le(s) effet(s) de bord.

  Remarque : penser à écrire le code de manière à faciliter l'écriture des tests (par ex. : fonctions d'E/S prenant en paramètre un `Writer` et non un `FileWriter` pour permettre l'écriture de tests utilisant un `StringWriter`).

- Comment tester une méthode protégée ?

  Placer la classe de test dans le même package que la classe testée.

Problèmes courants 2/2

- Comment tester une méthode privée ?

  Normalement c'est impossible, il est néanmoins possible de contourner le problème en utilisant le mécanisme de *reflection* pour outre-passer les droits d'accès[10].

- Comment tester une interface graphique ?

  Pas de solution idéale, les tests unitaires ne sont pas adaptés : soit tout tester à la main..., soit utiliser un « robot » (un simulateur programmable d'action sur l'UI)[11].

[10] Lire l'article suivant pour davantage de détails : `http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html`

[11] `http://abbot.sourceforge.net/`, par exemple.

(R)appel : importation de champs statiques (JDK $\geq$ 1.5)

La directive `import static` permet de rendre les champs statiques d'une classe directement accessibles (sans avoir à préfixer le nom de la classe).

Exemple : `import static java.lang.Math.*;`

Sans `import static` :

```
1     double x = Math.sqrt(2.0);
```

Avec `import static` :

```
1  import static java.lang.Math.*;
2  // ou import java.lang.Math.sqrt;
3      /* ... */
4      double x = sqrt(2.0);
```

(R)appel : annotation (JDK $\geq$ 1.5)

Syntaxe spéciale pour ajouter des méta-données au code source.

Il est possible d'annoter un package, une classe, un champ, un constructeur, une méthode, un paramètre, une variable locale.

```
1 public @interface MonAnnotation { /* ... */ }
2 // Declare une annotation MonAnnotation.
3
4 @MonAnnotation
5 public void uneMethode() { /* ... */ }
6 // @MonAnnotation est une annotation de
7 //      la methode uneMethode().
```

Ces méta-données sont disponibles à l'exécution via le mécanisme de *reflection* (à l'opposé des commentaires Javadoc, par exemple).

JUnit 4 (différences avec JUnit 3.x)

Importation : `junit.framework` → `org.junit`

Plus besoin d'hériter de `junit.framework.TestCase`.

Rajouter un `import static org.junit.Assert.*;` (ou faire au cas par cas).

Annotations (à importer également) :

- `@Before` pour marquer la méthode `setUp()` (qui doit être `public`),
- `@After` pour marquer la méthode `tearDown()` (qui doit être `public`),
- `@Test` pour marquer les méthodes de test (qui doit être `public`).

Remarque : il n'est plus nécessaire de respecter le nom des fonctions...

Dans Eclipse, sélectionner JUnit 4 pour la création de test unitaire ou l'ajout de la bibliothèque dans le *Build Path*.

Exemple (JUnit 3.x)

```
1  import java.util.ArrayList;
2  import java.util.Collection;
3
4  import junit.framework.Test;
5  import junit.framework.TestCase;
6
7  public class ArrayListTest3 extends junit.framework.TestCase {
8      private Collection collection;
9
10     protected void setUp() {
11         collection = new ArrayList();
12     }
13
14     public void testEmptyCollection() {
15         assertTrue(collection.isEmpty());
16     }
17
18     protected void tearDown() {
19         collection = null;
20     }
21 }
```

Exemple (JUnit 4)

```java
import java.util.ArrayList;
import java.util.Collection;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class ArrayListTest4 {
    private Collection collection;

    @Before
    public void setUp() {
        collection = new ArrayList();
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @After
    public void tearDown() {
        collection = null;
    }
}
```

Introduction

Définition

Un outil de couverture de code est un outil permettant de vérifier dynamiquement (à l'exécution) quelles parties du code source à tester ont effectivement été exécutées (« couvertes » par le(s) test(s)).

Il s'agit d'une aide/d'un complément essentiel pour l'écriture de tests unitaires.

Critères de couverture 1/2

Les principaux critères de couverture sont les suivants :

- *Function Coverage* : quelles fonctions/méthodes ont été exécutées ?
- *Statement Coverage* : quelles lignes du code source ont été exécutées ?
- *Branch Coverage* : quelles parties d'un bloc conditionnel (`if`/`else`) ont été exécutées ?
- *Condition Coverage* : quelles valeurs de décision (ex : `i < num && !skip`) ont été rencontrées à l'exécution ?
- *Entry/Exit Coverage* : toutes les possibilités d'invocation et de sortie d'une fonction/méthode ont-elles été utilisées ?
- *Loop Coverage* : aucune, une seule, plusieurs exécutions successives d'une boucle ?
- *Path Coverage* : tous les chemins d'exécution d'une partie du code ont-ils été empruntés ?

Critères de couverture 2/2

Statement Coverage $\not\Rightarrow$ *Condition Coverage* :

```
1  void m(int i) {
2      System.out.println("Hello");
3
4      if (i <= 0)
5          System.out.println(", World!");
6
7      System.out.println("!");
8  }
```

Si `m(-1)` est appelée dans un test unitaire, 100% de *Statement Coverage*, mais 50% de *Condition Coverage*.

Path Coverage $\Rightarrow$ *Statement* / *Condition* / *Branch* / *Entry* /*Exit* / *Loop Coverage*

Couverture de code dans Eclipse

Il n'y a pas (pour l'instant) d'outil de couverture de code intégré à Eclipse par défaut.

Plusieurs greffons propriétaires/commerciaux :

- Clover pour Eclipse[12],
- ...

Quelques greffons libres/gratuits :

- Coverlipse[13], projet inactif, basique mais fonctionne avec Eclipse 3.2,
- CodeCover[14], projet actif, plus évolué mais requiert Eclipse 3.3.

[12] `http://www.atlassian.com/software/clover/`
[13] `http://coverlipse.sf.net/`
[14] `http://codecover.org/`

Coverlipse

Pour l'installer, suivre les instructions à l'adresse :

`http://coverlipse.sourceforge.net/download.php`

Gère uniquement le *Statement coverage*.
Pour l'utiliser :

- sélectionner la classe de tests unitaires et la lancer comme *[JUnit w/ Coverlipse]* : sélectionner la classe dans la vue *Package Explorer* puis *[Mouse-R / Run as / Run... / JUnit w/ Coverlipse]*.
- dans la boîte de dialogue *Run*, sélectionner l'onglet *Package Filter* et ajouter des filtres pour inclure toutes les classes à utiliser.

Comment ça marche ? Coverlipse modifie le *bytecode* Java pour l'« instrumenter » et générer les informations de couverture de code.
En plus d'annoter le code dans la vue de l'éditeur Java, le greffon fournit plusieurs vues (*Coverlipse Class / Marker View*) pour consulter les informations de couvertures.

CodeCover

Pour l'installer, suivre les instructions à l'adresse :

`http://codecover.org/documentation/install.html`

Gère les critéres : *Statement Coverage*, *Branch Coverage*, *Condition Coverage* et *Loop Coverage*.
Pour l'utiliser :

- activer *CodeCover* pour le projet (aller dans les propriétés du projet), puis sélectionner les critères de couverture,
- choisir les classes à utiliser pour l'analyse (*[Mouse-R / Use For Coverage Measurement]*),
- sélectionner la classe de tests unitaires et la lancer comme *[CodeCoder Measurement For JUnit]* : sélectionner la classe dans la vue *package Explorer* puis *[Mouse-R / Run as / Run... / CodeCoder Measurement for JUnit]*.

Le greffon fournit une nouvelle perspective et plusieurs vues pour consulter les informations de couvertures.

Démonstration