Les tests logiciels : séance 4

M.Madani

ISIL - HEPL

2015-2016

Références

- Foundations of Software Testing, Aditya P. Mathur,Pearson Education India,2008.
- Software Testing and Quality Assurance,Kshirasagar Naik,Priyadarshi Tripathy,Wiley,2008.
- JUnit, Mise en oeuvre pour automatiser les tests en Java,Benoît Gantaume,ENI Editions.
- `http://xunitpatterns.com/`, Mars 2014.
- `http://martinfowler.com/articles/mocksArentStubs.html`,Mars 2014.
- `https://code.google.com/p/mockito/`,Mars 2014.

Tests unitaires

- En quelques mots
  - Se décline en tests statiques et tests dynamiques.
  - Ne correspond pas à une technique en particulier mais plutôt sur quoi le test s'applique.
- Hypothèses
  - Tester d'abord séparément les unités de code augmente la qualité au final du système.
  - Mettent en évidence plus facilement des fautes puisque l'unité est testée hors du contexte d'exécution final.
- Limitations
  - La mise en place des tests pour ne pas dépendre d'autres unités peut s'avérer compliqué.
  - Pour des tests statiques, les facteurs humains et de temps sont à prendre en considération.

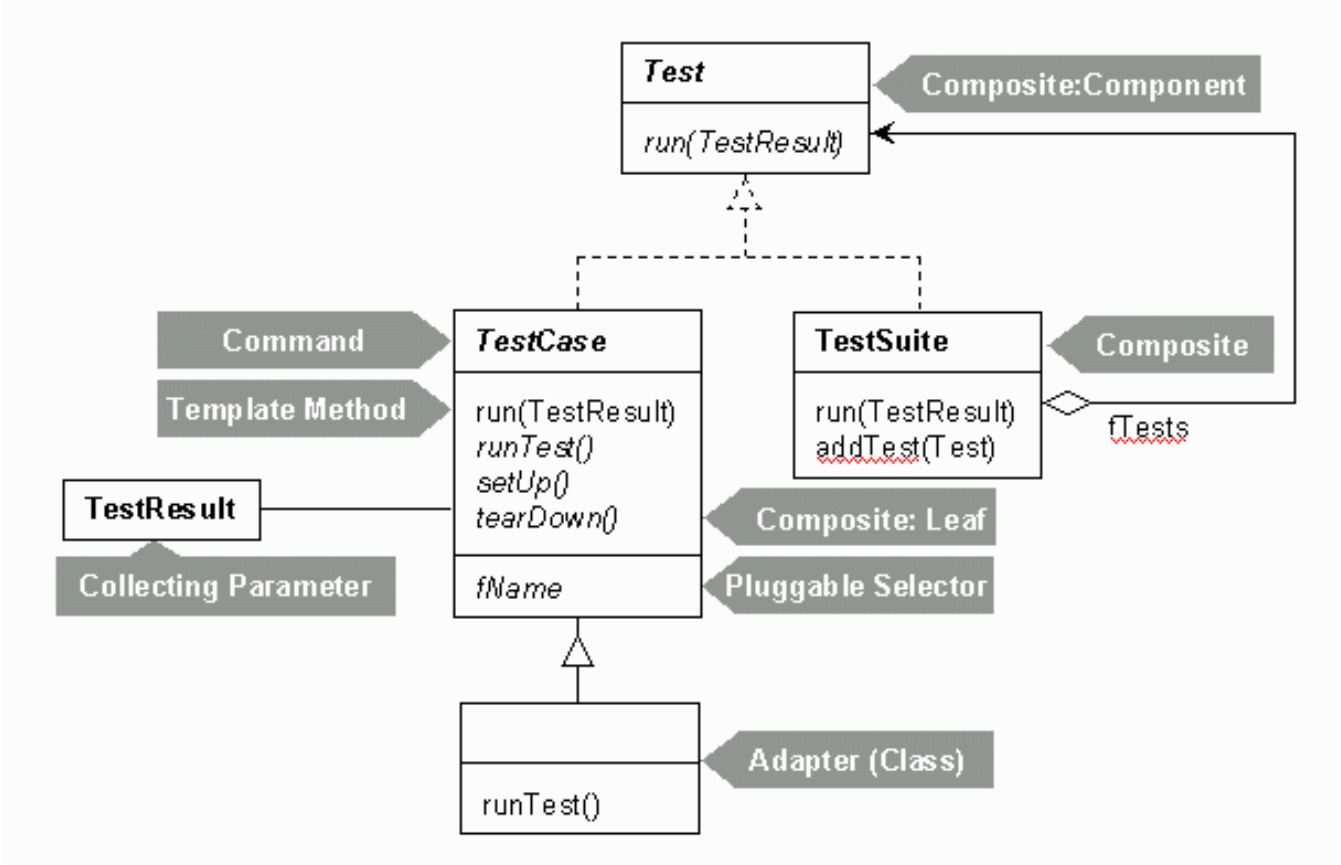
Unité de code

- Une unité de code a une responsabilité clairement identifiée et limitée.
- Cela peut être par exemple une fonction ou une classe.
- La création de l'unité et la vérification de son bon fonctionnement est généralement la responsabilité d'un seul développeur.
- Un grand nombre de techniques de tests sont disponibles pour tester les unités(EC, control flow testing, ...).

Environnement d'exécution des tests

- L'environnement d'exécution correspond à l'émulation du contexte d'exécution de l'unité testée.
- Le contexte de l'unité testée est constitué
  - d'un driver : un élément de code qui appelle l'unité testée(UUT -Unit Under Test).
  - Le remplacement des unités appelées depuis l'UUT à l'aide de doublures.

Framework de test



Source :http ://junit.sourceforge.net/doc/cookstour/cookstour.htm

Structure d'un test

- Structure générale
  - Mise en place : il est possible de partager une logique commune de mise en place entre plusieurs tests.
  - Le test proprement dit : chaque test est exécuté de manière isolée des autres.
  - Le nettoyage.
- Exemple

```
1 public class ATest
2 {
3   public ATest () { }
4   @Before public void setUp (){ }
5   @After public void tearDown (){ }
6   @Test public void testMethod (){ }
7 }
```

Les doublures

- Dummy : objet vide juste pour satisfaire une invocation.
- Fake : objet ayant une implémentation simplifié et destiné à satisfaire pour un test.
- Stub : s'écartant plus du véritable objet qu'un fake, il permet le contrôle des entrées indirecte de l'UUT.
- Mock : permet de fournir les entrées indirectes de l'UUT et la vérification des sorties indirectes.
- Spy : proxy vers le véritable objet, il permet d'enregistrer les sorties indirectes l'UUT a effectué. Certaines méthodes peuvent être "stubbées".

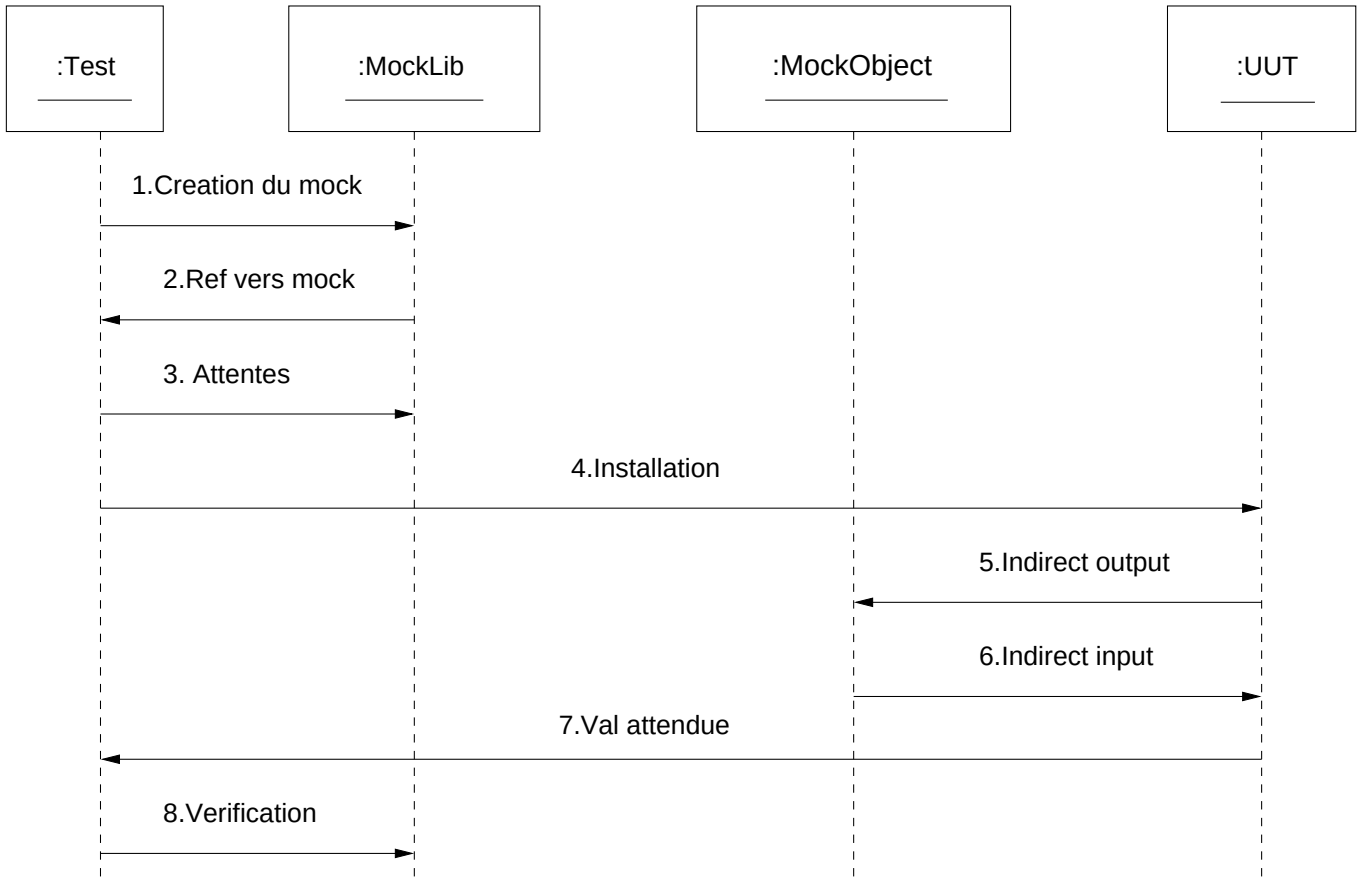
Pourquoi utiliser des doublures

- Le sous système est trop lent.
- Le sous système ne peut être automatisé : i.e. demande d'une entrée utilisateur.
- L'objet réel n'existe pas encore.
- Le sous système est trop complexe à mettre en place : i.e. en termes de déploiement physique.
- Effet réel non souhaité : i.e. on peut vouloir préféré un affichage à la création d'un fichier sur disque.
- etc...

Mise en place d'une doublure

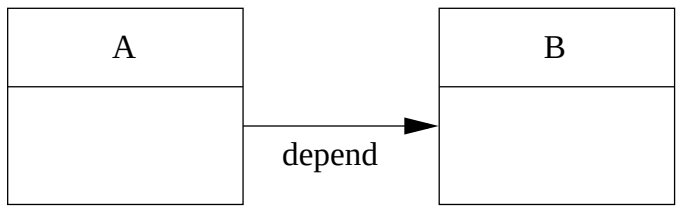
- L'unité testée
  - permet d'injecter l'unité à doubler. L'environnement d'exécution la crée et l'injecte.
  - peut être modifiée : inversion de dépendances et injection de la dépendance.
  - ne peut être modifiée : si l'unité est une classe, alors la rendre testable nécessite de trouver une solution :
    - Outre passer les clauses d'accès.
    - Créer une classe fille permettant l'injection.

Mise en place d'un mock



Inversion des dépendances

- A dépend de B $\Rightarrow$ difficile de réutiliser A dans d'autres contextes.



Inversion des dépendances

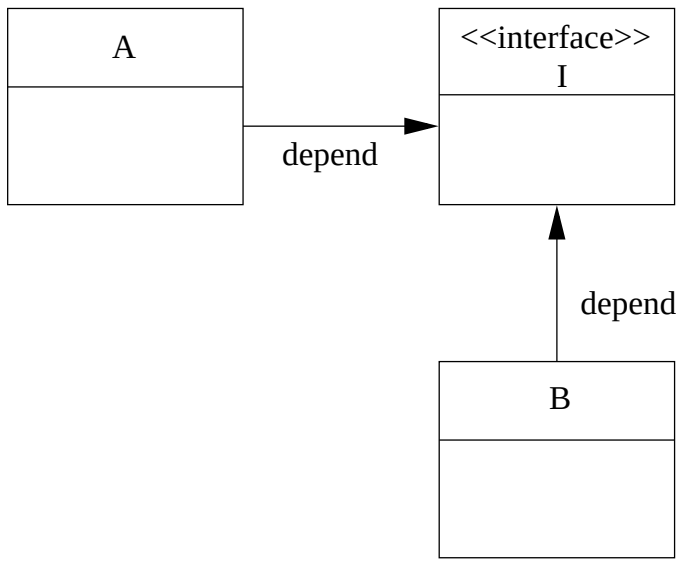
```
1  public class A
2  {
3    private B b = new B ();
4    public void callB ()
5    {
6      b.doX();
7    }
8    ...
9      new A ().callB ();
10   ...
11 }
```

```
1  public class B
2  {
3
4    public void doX ()
5    {
6     //time consuming task
           ...
7    }
8
9  }
```

Inversion des dépendances

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux devraient dépendre d'abstractions.
- Les abstractions ne devraient pas dépendre de détails. Les détails devraient dépendre d'abstractions.

- Inversion des dépendances. Le bas niveau se conforme à une interface utilisée par le haut niveau.



Inversion des dépendances

```
 1 public class A
 2 {
 3   private I i;
 4   public A (I o)
 5   {
 6     i = o;
 7   }
 8   public void callMethodOnB ()
 9   {
10     i.doX ();
11   }
12   ...
13     new A (new B ()).callMethodOnB ();
14     new A (new I () {
15    public void doX ()
16    {
17    System.out.println ("doX");
18    }
19    }).callMethodOnB ();
20 }
```

Inversion des dépendances

```
1 public class B implements I
2 {
3   public void doX ()
4   {
5     System.out.println ("doX");
6   }
7 }
```

```
1 interface I
2 {
3   void doX ();
4 }
```

Mock et Stub

- Les deux termes désignent une doublure pour l'unité à remplacer.
- Un stub fournit des résultats aux appels réalisés dessus lors d'un test.
  - Vérification basée sur un état : on vérifie que l'UUT est dans l'état attendu à la fin du test.
  - Un stub ne fait pas échouer le test.
- Les mocks sont des objets sur lesquels on spécifie les appels qu'ils sont sensés recevoir.
  - Test de comportements : les appels effectués depuis l'UUT vers l'extérieur sont spécifiés et vérifiés.
  - Les tests ne sont plus qualifiés en boîte noire.
  - L'utilisation de doublures mocks dans un processus TDD permet d'élaborer la spécification des unités dont dépend l'unité testée.

Quelques possibilités des mocks

- Tester si une méthode a été appellée ou non.
- Tester l'ordre d'appels de plusieurs méthodes.
- Vérifier avec un certain laps de temps.
- Tester le nombre d'invocations d'une méthode.
- Utiliser des comparateurs sur les arguments(matchers).
- etc...

Exemple d'utilisation d'une doublure

```
1  public class ATest
2  {
3    A a;
4    B b;
5     @Before public void setUp ()
6    {
7      b = mock (B.class);
8      a = new A (b);
9    }
10   @Test public void testCallDoX ()
11   {
12     a.callDoX ();
13     verify (b).doX ();
14   }
15   @Test public void testCallDoY ()
16   {
17     stub (b.getY ()).toReturn (-1);
18     assertEquals (-1, a.callDoY ());
19   }
20 }
```

La classe A

```
1  public class A
2  {
3    private B b;
4    public A (B o)
5    {
6      b = o;
7    }
8    public void callDoX ()
9    {
10     b.doX ();
11   }
12   public int callDoY ()
13   {
14     return b.getY ();
15   }
16 }
```

La classe B

```
 1 public class B
 2 {
 3   void doX ()
 4   {
 5     System.out.println ("doX");
 6   }
 7   int getY ()
 8   {
 9     return 0;
10   }
11 }
```