ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Doublures dans les tests

Introduction à Mockito

1. Introduction

Mockito est un framework open source de programmation de doublures pour le langage de programmation Java. Mockito est utilisé pour simuler des interfaces afin qu'une fonctionnalité fictive puisse remplacer une fonctionnalité réelle que l'on ne souhaite pas utiliser dans un test unitaire.

Des alternatives à Mockito sont par exemple jMock ou encore EasyMock.

Mockito n'est pas nativement présent dans les IDE. Il doit être associé par l'intermédiaire d'un fichier jar. Le framework Mockito est hébergé à l'adresse : http://site.mockito.org/.

1.1 Installation sous Netbeans

Le framework Mockito est hébergé à l'adresse :http://site.mockito.org. L'installation sous Netbeans se fait par un lien de bibliothèque au projet :

- Ouvrir la fenêtre de propriétés du projet.
- Sélectionner l'onglet « Libraries » puis « Compile test ».
- Cliquer sur « Add jar/folder ».
- Sélectionner le fichier jar du dossier de Mockito.

2. Tests unitaires et doublures

Un test unitaire teste une classe en isolation. Les effets de bord des autres classes ou du système doivent être éliminés si possible pendant le test. L'isolation permet

ainsi de circonscrire les erreurs uniquement dans la classe en test.

L'exemple classique est celui du test d'une classe qui fait appel à une base de données. L'utilisation d'une base de données réelle rend l'écriture des tests très complexe, leur exécution très longue, et si la base de données est sur le réseau, très dépendante des conditions de test.

C'est là qu'interviennent les doublures. Le mot doublure reprend l'acception de doublure du cinéma. Il envoie aussi la notion de bouchon. Une doublure est un objet qui simule le comportement d'une classe et garantie que les conditions du test sont toujours les mêmes. Ces objets doublures sont fournis à la classe à tester et remplacent les objets réels.

Les doublures permettent l'écriture de test pour une classe alors que les classes dont elle dépend ne sont pas encore développées ou que leur exécution est trop longue ou qu'elles ont un comportement correct mais imprévisible.

3. Mockito

Mockito est un générateur automatique de doublures qui peut être utilisé en conjonction avec JUnit. Il permet de créer des objets doublures à partir de n'importe quelle classe ou d'interface. Les doublures sont totalement contrôlables. Il suffit d'en déterminer le comportement en imposant les valeurs de retour aux méthodes : c'est qu'on appelle le "stubbing".

Mockito offre aussi un mécanisme d'espionnage qui donne accès aux statistiques d'utilisation d'objet dans les tests.

3.1 Cycle de vie d'un mock dans un cas de test

La procédure d'utilisation de Mockito est très simple :

1. Création d'un objet mock pour une classe ou une interface ;
2. Description du comportement qu'il est censé imiter ;
3. Utilisation du mock dans le code de test ;
4. Si nécessaire, interrogation du mock pour savoir comment il a été utilisé durant le test.

Dans la suite, nous présentons d'abord un exemple récapitulatif succinct

puis détaillons chacune des étapes de l'utilisation de doublure.

3.2 Exemple complet

Prenons l'exemple de l'interface `List` de l'API Java SE que l'on veut doubler pour tester une classe qui manipule cette liste sans se préoccuper de gérer effectivement cette liste.

3.2.1 Création de doublure

Une doublure est créée à partir de l'interface `List`.

```
List mockedList = mock(List.class);
```

Pour l'instant son comportement est celui par défaut. Un appel à `mockedList.get(0)` retournera `null`.

La classe à tester se nomme `MaClasse` et possède un attribut qui est du type `List` que l'on veut isoler En bon code testable, le constructeur prend le type de liste effectif comme paramètre, ce qui permet de remplacer la liste en attribut de la classe par la doublure.

```
MaClasse c = new MaClasse(mockledList);
```

Pour contrôler le comportement de la doublure que nous avons créée, nous allons faire du "stubbing" en fonction des appels que l'on veut simuler.

3.2.2 Le "stubbing"

La stubbing consiste à définir le comportement souhaité pour chaque appel des méthodes de l'objet doublé. Supposons que pour tester la classe `MaClasse` nous appelons la méthode `get(0)` de la doublure et que la réponse attendue soit la chaîne "toto". Dans ce cas, il faut définir le comportement de cet appel particulier :

```
when(mockedList.get(0)).thenReturn("toto");
```

Si l'on souhaite répondre "toto" pour n'importe qu'elle indice de la liste :

```
when(mockedList.get(anyInt())).thenReturn("toto");
```

3.2.3 Utilisation des doublures

La ligne suivante s'assure que la valeur retournée par `get(10)` est bien "toto". À défaut, une exception est levée.

```
assertEquals("Problème d'insertion", mockedList.get(10), "toto");
```

Il peut être intéressant de contrôler qu'une méthode a bien été appelée et avec les bons paramètres.Si l'on souhaite vérifier qu'il y a bien eu un appel à la méthode `get` avec comme paramètre 10. À défaut, la JVM lève une exception :

```
verify(mockedList).get(10);
```

4. Création de doublure

L'utilisation de Mockito nécessite au préalable l'importation statique du paquet `Mockito` :

```
import static org.mockito.Mockito.*;
```

Il y a ensuite deux manières de créer des doublures, avec la méthode `mock()` et avec l'annotation `@Mock`.

4.1 Création d'une doublure avec la méthode mock()

C'est la méthode la plus directe. Par exemple pour créer une doublure de la classe `MaClasse` et l'injecter dans un objet `Foo` :

```
   public void FooTest {
     private Foo _foo;

     @Before
     public void setUp() {
        MaClasse monMock = Mockito.mock(MaClasse.class);
        _foo = new Foo(monMock);
     }
   ...// utilisation de foo dans les méthodes de test.
   }
```

La même création avec un nom rend les messages d'erreur plus clairs :

```
MaClasse mockAvecNom = mock(MaClasse.class, "Mon mock");
```

4.2 Création d'une doublure avec l'annotation @mock

L'annotation se met au-dessus de la déclaration de l'attribut du type du mock. Le nom du mock est automatiquement celui de l'attribut. Toutefois, il ne faut pas oublier d'initialiser l'interpréteur des annotations de Mockito. Pour cela,

il y a trois façons de faire : par l'appel explicite de la fonction d'initialisation, par l'utilisation d'un moteur d'exécution JUnit ou par l'appel d'une règle JUnit.

4.2.1 Initialisation par l'appel de méthode initMock()

```
public void FooTest {
  private Foo _foo;
  @Mock
  private MaClasse _monMock;

  @Before
  public void setUp() {
    MockitoAnnotations.initMocks();
    _foo = new Foo(_monMock);
  }
...// utilisation de foo dans les méthodes de test.
}
```

4.2.2 Initialisation par le moteur d'exécution MockitoJUnitRunner

Cette option n'est utilisable que si Mockoto est le seul moteur utilisé. Ceci exclut par exemple l'utilisation des paramètres de test JUnit.

```
@RunWith(MockitoJUnitRunner.class)
public void FooTest {
  private Foo _foo;
  @Mock
  private MaClasse _monMock;

  @Before
  public void setUp() {
    _foo = new Foo(_monMock);
  }
...// utilisation de foo dans les méthodes de test.
}
```

4.3 Initialisation par la règle MockitoRule

La règle JUnit invoque implicitement la méthode `initMocks(this)` :

```
public void FooTest {
  private Foo _foo;
  @Mock
  private MaClasse _monMock;
  @Rule
  public MockitoRule mockitoRule = MockitoJUnit.rule();

  @Before
  public void setUp() {
    _foo = new Foo(_monMock);
  }
...// utilisation de foo dans les méthodes de test.
}
```

5. Le stubbing

Littéralement *stub* signifie bouchon. Le stubbing consiste à définir le comportement des méthodes d'un mock. Attention, il est impératif que la classe et les méthodes à doubler ne soient pas `final`.

5.1 Appel d'une méthode

5.1.1 Appel de méthode sans paramètre avec une valeur de retour unique

Le stubbing :

```
when(monMock.retourneUnEntier()).thenReturn(3);
```

et le test JUnit :

```
assertEquals("Premièr appel ", 3, monMock.retourneUnEntier());
assertEquals("Deuxième appel ", 3, monMock.retourneUnEntier());
```

Retourne la valeur stubbée autant de fois que nécessaire.

5.1.2 Appel de méthode sans paramètre avec des valeurs de retour consécutives

Le stubbing :

```
when(monMock.retourneUnEntier()).thenReturn(3, 4);
```

qui peut aussi s'écrire avec un *Builder* :

```
when(monMock.retourneUnEntier()).thenReturn(3).thenReturn(4);
```

et le test JUnit :

```
assertEquals("Premier appel : 3", 3, monMock.retourneUnEntier());
assertEquals("Deuxième appel : 4", 4, monMock.retourneUnEntier());
```

5.1.3 Appel de méthode avec utilisation de la valeur des paramètres

Soit la méthode :

```
public int retourneUnEntierBis(int i, int j);
```

Le stubbing :

```
when(monMock.retourneUnEntierBis(4, 2)).thenReturn(1);
when(monMock.retourneUnEntierBis(5, 3)).thenReturn(2);
```

et l'utilisation avec JUnit :

```
assertEquals("Appel avec 4 2: 1", 1, monMock.retourneUnEntierBis(4, 2));
assertEquals("Appel avec 5 3: 2", 2, monMock.retourneUnEntierBis(5, 3));
```

5.1.4 Appel de méthode avec n'importe quel paramètre

Souvent, on veut spécifier un appel sans que les valeurs des paramètres aient vraiment d'importance. On utilise pour cela des *Matchers*.

```
import org.mockito.Matchers;
when(mockedList.get(anyInt())).thenReturn("element");
```

Voici une liste des matchers classiques :

- `any(), anyObject().`
- `anyBoolean(), anyDouble(), anyFloat() , anyInt(), anyString() …`
- `anyList(), anyMap(), anyCollection(), anyCollectionOf(), anySet(), anySetOf().`

Attention ! Si on utilise des Matchers, tous les arguments doivent être des Matchers. La ligne suivante produit une erreur de compilation :

```
when(mock.someMethod(anyInt(), "une valeur")).thenReturn("element");
```

5.2 Levée d'exception

On suppose donnée la méthode suivante qui lève l'exception `BidonException` :

```
public int retourneUnEntierOuLeveUneException() throws BidonException;
```

Le stubbing est :

```
when(monMock.retourneUnEntierOuLeveUneException()).thenThrow(new
BidonException());
```

ou une autre écriture :

```
doThrow(new
BidonException()).when(monMock).retourneUnEntierOuLeveUneException();
```

et le test JUnit :

```
@Test(expected = BidonException.class)
public void should_throw_exception() {
     monMock.retourneUnEntierOuLeveUneException();
}
```

Il est possible de cumuler le retour d'une valeur donnée puis la levée d'une exception :

```
when(monMock.retourneUnEntierOuLeveUneException()).thenReturn(3).thenThrow(n
ew BidonException());
```

5.3 Limitations

Mockito possède certaines limitations. Premièrement, il ne peut pas tester les constructions suivantes :

- Les classes déclarées `final`.
- Les classes anonymes.
- Les types primitifs.

Deuxièmement, il ne peut pas stubber les méthodes suivantes :

- `equals()`. Mockito vérifie les valeurs en arguments en utilisant `equals()`.
- `hashCode()`.

6. Espionnage

Mockito garde trace de tous les appels de méthode avec leurs valeurs de paramètre. Vous pouvez utiliser la méthode `verify()` sur un objet mock pour vérifier qu'une méthode a été appelée avec certaines valeurs de paramètres. Ce type de test correspond à un *test de comportement*, parce qu'il ne teste pas le résultat de l'appel, mais la façon dont il a été appelé.

```
   @Test
   public void test1() {
      MyClass test = Mockito.mock(MyClass.class);
      // Définit la valeur de retour pour la méthode getUniqueId()
      test.when(test.getUniqueId()).thenReturn(43);

      // Le test utilisant le mock de MyClass

      // Verifie que la méthode a été appelée avec le paramètre 12
      Mockito.verify(test).testing(Matchers.eq(12));

      // Vérifie que la méthode a été appelée deux fois
      Mockito.verify(test, Mockito.times(2));
}
```

6.1 Fonctionnement de la méthode verify()

Elle permet de vérifier :

- quelles méthodes ont été appelées sur un mock,
- combien de fois,
- avec quels paramètres,

- dans que l'ordre.

Si la vérification échoue, il y a levée d'exception et le test unitaire échoue.

6.2 Vérification du nombre d'appels

Il est possible de vérifier qu'une méthode, par exemple `retourneUnBooleen`, doit avoir été appelée :

- exactement une fois :

```
verify(monMock).retourneUnBooleen();
```

ou

```
verify(monMock, times(1)).retourneUnBooleen();
```

- au moins une fois :

```
verify(monMock, atLeastOnce()).retourneUnBooleen();
```

- au plus une fois :

```
verify(monMock, atMost(1)).retourneUnBooleen();
```

- jamais :

```
verify(monMock, never()).retourneUnBooleen();
```

6.3 Vérification des paramètres d'appel

On peut vérifier que `retourneUnEntierBis` a bien été appelée avec les paramètres 4 et 2 en utilisant l'instruction :

```
verify(monMock).retourneUnEntierBis(4, 2);
```

6.4 Vérification de l'ordre des appels

Cette vérification nécessite d'importer la classe `InOrder`.

```
import org.mockito.InOrder;
```

Pour vérifier que l'appel (4, 2) est effectué avant l'appel (5, 3) :

```
InOrder ordre = inOrder(monMock);
ordre.verify(monMock).retourneUnEntierBis(4, 2);
ordre.verify(monMock).retourneUnEntierBis(5, 3);
```

Fonctionne aussi avec plusieurs mocks :

```
InOrder ordre = inOrder(mock1, mock2);
ordre.verify(mock1).foo();
ordre.verify(mock2).yo();
```

6.5 Appel avec n'importe quel paramètre

On utilise pour cela des *Matchers* déjà présentés en section 5.1.4

```
verify(mockedList).get(anyInt());
```

Rappel : si on utilise des Matchers, tous les arguments doivent être des Matchers. La ligne suivante produit une erreur de compilation :

```
verify(mock).someMethod(anyInt(), anyString(), "third argument");
```

6.6 Et aussi...

La méthode `verifyNoMoreInteraction()` permet de vérifier qu'aucune autre méthode de l'objet n'a été appelée.

La méthode `verifyZeroInteraction(mockTwo, mockThree)` vérifie que d'autres objets mocks n'ont pas été appelés après cette instruction.

6.7 Espionner les méthodes classiques

Il est aussi possible d'espionner un objet classique, qui n'est pas un objet mock. L'annotation `@Spy` ou la méthode `spy()` peuvent être utilisées pour espionner un objet concret.

```
List list = new LinkedList();
List spy = spy(list);

// Appel d'une méthode
verify(spy).add("one");

// Vérifie l'effectivité de l'appel
verify(spy).add("one");
```

7. Utilisation de l'annotation @InjectMocks

Imaginons que nous ayons deux classes, l'une dépendant de l'autre :

```
public class Another {
   public final void doSomething() {
      System.out.println("Un autre service fonctionne...");
   }
}
public class One {
   private final transient Another _another;

   public final void work() {
      System.out.println("Un premier service fonctionne");
      _another.doSomething();
   }
```

```
}
```

Maintenant, si nous voulons tester `One`, nous avons besoin d'une doublure de la classe `Another`. Une approche classique serait de passer la doublure `Another` dans le constructeur de `One` :

OneTest.java :

```
public final class OneTest {
   @Mock
   private transient Another _another;
   private transient One _one;

   @Before
   public void setup() {
      MockitoAnnotations.initMocks(this);
      _one = new One(_another);
   }

   @Test
   public void oneCanWork() throws Exception {
      _one.work();
      Mockito.verify(_another).doSomething();
   }
}
```

Cela fonctionne, mais il n'est pas nécessaire d'appeler le constructeur de `One` explicitement, nous utilisons simplement `@InjectMocks` :

OneTest.java (2) :

```
public final class OneTest {
   @Mock
   private transient Another _another;
   @InjectMocks
   private transient One _one;

   @Before
   public void setup() {
      MockitoAnnotations.initMocks(this);
   }

   @Test
   public void oneCanWork() throws Exception {
      one.work();
      Mockito.verify(_another).doSomething();
   }
}
```

Il y a quelques limitations à l'utilisation, mais pour la plupart des cas cela fonctionne parfaitement.