ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Doublures dans les tests Java

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. Dans un projet Gradle, il faut ajouter les lignes suivantes dans le fichier `build.gradle` :

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:2.+" }
```

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'exécution des tests 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 du mot doublure dans le cinéma. Il renvoie aussi à 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 : 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.

La section suivante donne un exemple récapitulatif de l'utilisation de Mockito.

3.2 Exemple récapitulatif

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 la méthode `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 la remplacer par une 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 JUnit suivante s'assure que la valeur retournée par `get(10)` est bien "toto".

```
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.

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

4. Création de doublure

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

```
1 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 Limitation

Mockito ne peut pas doubler les classes déclarées `final`.

4.2 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` :

```
1   public void FooTest {
2      private Foo _foo;
3
4      @Before
5      public void setUp() {
6         MaClasse monMock = Mockito.mock(MaClasse.class);
7         _foo = new Foo(monMock);
8      }
9      ... // utilisation de foo dans les méthodes de test.
10  }
```

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

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

4.3 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.

4.3.1 Initialisation par l'appel de méthode initMocks()

C'est la méthode la plus classique.

```
1    public void FooTest {
2       private Foo _foo;
3       @Mock
4       private MaClasse _monMock;
5
6       @Before
7       public void setUp() {
8          MockitoAnnotations.initMocks(this);
9          _foo = new Foo(_monMock);
10      }
11      ... // Les méthodes de test utilisant _foo et _monMock.
12   }
```

La méthode `MockitoAnnotations.initMocks(this)` initialise tous les attributs annotés par `@Mocks`, qui se trouvent dans la classe passée en paramètre.

4.3.2 Initialisation par le moteur d'exécution MockitoJUnitRunner

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

```
1    @RunWith(MockitoJUnitRunner.class)
2    public void FooTest {
3       private Foo _foo;
4       @Mock
5       private MaClasse _monMock;
6
7       @Before
8       public void setUp() {
9          _foo = new Foo(_monMock);
10      }
11      ... // Les méthodes de test utilisant _foo et _monMock.
12   }
```

4.3.3 Initialisation par la règle MockitoRule

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

```
1    public void FooTest {
2       private Foo _foo;
3       @Mock
4       private MaClasse _monMock;
5       @Rule
```

```
6       public MockitoRule mockitoRule = MockitoJUnit.rule();
7
8       @Before
9       public void setUp() {
10         _foo = new Foo(_monMock);
11      }
12      ... // Les méthodes de test utilisant _foo et _monMock.
13  }
```

5. Le stubbing

Littéralement *stub* signifie bouchon. Le stubbing consiste à définir le comportement des méthodes d'un mock.

5.1 Limitations

Attention, il est impératif que les méthodes à doubler ne soient pas `final`.

De plus, Mockito ne peut pas doubler les deux méthodes suivantes :

- `equals()`.
- `HashCode()`.

La raison est que Mockito utilise ces méthodes en interne.

5.2 Appel d'une méthode

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

Le stubbing :

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

et un test JUnit vérifiant que la méthode retourne la valeur à chaque appel :

```
1 assertEquals("Premier appel ", 3, monMock.retourneUnEntier());
2 assertEquals("Deuxième appel ", 3, monMock.retourneUnEntier());
```

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

Le stubbing :

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

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

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

et un test JUnit vérifiant que la méthode retourne les bonnes valeurs successivement :

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

5.2.3 Appel de méthode avec utilisation de la valeur des paramètres

Soit la méthode :

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

Le stubbing :

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

et un test JUnit vérifiant la bonne valeur de retour en fonction des valeurs de paramètre :

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

5.2.4 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*.

```
1 import org.mockito.Matchers;
2 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");
```

5.3 Levée d'exception

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

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

Le stubbing de cette méthode est :

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

ou une autre écriture :

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

et un test JUnit vérifiant que l'exception est bien levée à l'appel de la méthode :

```
1     @Test(expected = BidonException.class)
2     public void should_throw_exception() {
3           monMock.retourneUnEntierOuLeveUneException();
4     }
```

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

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

6. Espionnage

Mockito garde trace de tous les appels de méthode. Vous pouvez utiliser la méthode `verify()` sur un objet mock pour vérifier qu'une méthode a été appelée et avec certaines valeurs de paramètre. 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
   verify(test).testing(Matchers.eq(12));

   // Vérifie que la méthode a été appelée deux fois
   verify(test, 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 quel ordre.

Si la vérification échoue, il y a une 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 :

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

- au plus une fois :

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

- jamais :

```
1 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 :

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

6.4 Vérification de l'ordre des appels

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

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

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

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

Fonctionne aussi avec plusieurs mocks :

```
1 InOrder ordre = inOrder(mock1, mock2);
2 ordre.verify(mock1).foo();
3 ordre.verify(mock2).bar();
```

6.5 Vérification des appels avec n'importe quel paramètre

On utilise pour cela des *Matchers* déjà présentés en section 5.2.4.

```
1 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 :

```
1 verify(mock).someMethod(anyInt(), anyString(), "un argument effectif");
```

6.6 Vérification du non-appel d'une méthode

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. La méthode `spy()` peut être utilisée pour espionner un objet concret.

```
1    List<String> list = new LinkedList<>();
2    List spyList = spy(list);
3    @Test
4    public void test3() {
5       spyList.add("one");            // Appel d'une méthode
6       verify(spyList).add("one");    // Vérifie l'effectivité de l'appel
7    }
```

L'annotation `@Spy` remplace la méthode `spy()`. Toutefois, comme pour toutes les annotations Mockito, il faut autoriser les annotations avec l'instruction `initMocks()` ou l'utilisation du moteur `MockitoJUnitRunner` ou la règle `mockitoRule` (voir Sections 4.3.1, 4.3.2 et 4.3.3) :

```
1    @Spy
2    List<String> spyList = new LinkedList<>();
3    @Before
4    public void setUp() {
5       MockitoAnnotations.initMocks(this);
6    }
7    @Test
8    public void test3() {
9       spyList.add("one");            // Appel d'une méthode
10      verify(spyList).add("one");    // Vérifie l'effectivité de l'appel
11   }
```

7. Utilisation de l'annotation @InjectMocks

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

```
1    public class Two {
2       public final void doSomething() {
3          System.out.println("Un autre service fonctionne...");
4       }
5    }
6    public class One {
7       private final transient Two _two;
8       public One( Two two ) {
9          _two = two;
10      }
11      public final void work() {
12         System.out.println("Un premier service fonctionne");
13         _two.doSomething();
14      }
15   }
```

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

```
1    public final class OneTest {
2       @Mock
3       private Two _two;
4       private One _one;
```

```
5
6       @Before
7       public void setup() {
8          MockitoAnnotations.initMocks(this);
9          _one = new One(_two);
10      }
11
12      @Test
13      public void oneCanWork() throws Exception {
14         _one.work();
15         Mockito.verify(_another).doSomething();
16      }
17   }
```

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

```
1    public final class OneTest {
2       @Mock
3       private Two _two;
4       @InjectMocks
5       private One _one;
6
7       @Before
8       public void setup() {
9          MockitoAnnotations.initMocks(this);
10      }
11
12      @Test
13      public void oneCanWork() throws Exception {
14         one.work();
15         Mockito.verify(_another).doSomething();
16      }
17   }
```

Il y a quelques limitations à l'utilisation, mais pour la plupart des cas cela fonctionne parfaitement. L'intérêt apparaît surtout à ceux qui utilisent le framework Spring, ou les liens entre classes sont décrits par des annotations type `@Resource`.