

Doublures de test

Introduction

Les doublures de tests permettent d'isoler les classes à tester et de briser les interactions entre elles. Les doublures de tests ne remplacent pas JUnit et permettent de tester les appels que la classe va faire aux autres classes.

Par exemple, imaginons que l'on a une classe `Messagerie` qui peut lire les messages d'un·e utilisateur·ice sur base de son identifiant et mot de passe. Pour vérifier l'utilisateur·ice, la `Messagerie` va faire appel à la classe `Identification`. Cependant ici on ne veut tester uniquement `Messagerie` mais pas la classe d'`Identification`.

Pour avoir plus d'explications vous pouvez aller voir [cet article](#) (le code en exemple n'y est pas forcément de la meilleure des qualités mais c'est pratique pour comprendre le principe des doublures).

Pourquoi utiliser une doublure ?

Il y a plusieurs raisons pour lesquelles on voudrait mettre en place une doublure :

- La classe testée fait appel à un composant difficile ou coûteux à mettre en place (par exemple une base de données, c'est notamment le cas pour notre classe `Identification` dans l'exemple)
- Le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle (imaginons, une déconnexion réseau, on ne va pas réellement déconnecter la machine juste pour faire un test)
- La classe testée fait appel à un composant qui n'existe pas encore ou qui n'est pas suffisamment stable (cela permet par exemple de tester notre classe `Messagerie` alors que la classe `Identification` dont elle dépend n'existe pas encore)
- Le test fait appel à du code lent (par exemple, si notre `Identification` prends un certain temps, cela ralentirait grandement les tests pour rien)
- Le test fait appel à du code non déterministe (par exemple à l'heure ou à l'aléatoire. Par exemple si une classe ferait appel à une classe générant des nombres entre 1 et 6, nos tests seraient faux 5 fois sur 6)
- Séparer le code de test du code de l'application (par exemple si on crée une méthode `fakeGenerateNumber()` dans une classe aléatoire, cela complique les choses pour rien)

Types de doublures

Les stub objects

Un stub est simplement une classe écrite à la main spécialement pour le contexte du test. On sait quelle valeurs sont attendues d'avance et on les hardcode dans la classe.

Par exemple, dans le cas de l'identification de et de la messagerie on peut avoir une interface `Identification` et créer une classe `IdentificationStub` implémentant cette interface de façon à hardcoder les valeurs attendues pour le test (par exemple) :

```
public class IdentificationStub implements Identification {  
    boolean identify(String username, String password) {  
        // Renvois true si l'identifiant est 'toto' et le mot de passe 'mdp'  
        return "toto".equals(username) && "mdp".equals(password)  
    }  
}
```

Pour le test il suffit alors simplement d'injecter la classe `IdentificationStub` dans le constructeur de la classe `Messagerie`.

```
public class MessagerieTest {  
    @Test  
    void testLireMessages() {  
        // On injecte le stub dans la classe à tester  
        Messagerie messagerie = new Messagerie(new IdentificationStub());  
  
        // On fait les tests comme on le souhaite dessus...  
    }  
}
```

Les fake objects

Un fake est une doublure écrite à la main qui implémente le comportement attendu mais de façon plus simple que la classe réelle. Contrairement au stub qui est écrit spécialement pour un test précis, le fake a vocation à être suffisamment générique pour être utilisé dans plusieurs tests. Il est donc plus complexe que le stub mais plus réutilisable.

Pour reprendre l'exemple précédent on peut imaginer une classe `IdentificationFake` qui implémente `Identification` mais qui a une méthode supplémentaire `addAccount(String username, String password)` permettant de personnaliser le test.

```
public class IdentificationFake implements Identification {
    Map<String, String> comptes = new HashMap<String, String>();

    @Override
    public boolean identify(String username, String password) {
        // Vérifie que l'identifiant et le mot de passe soit dans la liste des comptes
        return comptes.containsKey(identifiant) && comptes.get(identifiant).equals(password);
    }

    public void addAccount(String username, String password) {
        // Ajoute un nouvel identifiant-mot de passe dans la fausse liste des comptes
        comptes.put(username, password);
    }
}
```

Comme pour le stub on peut donc aller l'injecter dans le constructeur lors du test, à la différence qu'ici on peut l'utiliser pour plusieurs tests différents et le configurer différemment à chaque fois.

```
public class MessagerieTest {
    private Identification identification = new IdentificationFake();

    @Test
    void testLireMessages() {
        // On configure notre fake object
        identification.addAccount("toto", "mdp");

        // On peut ensuite l'injecter dans notre classe à tester
        Messagerie messagerie = new Messagerie(identification);

        // Enfin on peut faire nos tests comme on le souhaite...
    }

    // on peut ensuite faire d'autres tests sur le même principe sans avoir à créer plusieurs classes pour chaque
    cas
}
```

Les dummy objects

Les dummy sont le type de doublure le plus simple, ce sont simplement des classes implémentant l'interface attendue mais ne faisant absolument rien car ils ne sont jamais vraiment utilisés.

Par exemple si on teste un cas précis où l'identification n'est jamais utilisée on peut créer une classe dummy implémentant `Identification` et qui renverrai toujours la même valeur (car quelque soit la valeur on s'en fout puis ce qu'elle ne sera pas utilisée) :

```
public class IdentificationDummy implements Identification {  
    @Override  
    public boolean identify(String username, String password) {  
        return true;  
    }  
}
```

Ici le code du test se fait exactement comme pour le stub

```
public class MessagerieTest {  
    @Test  
    void testLireMessages() {  
        // On injecte le stub dans la classe à tester  
        Messagerie messagerie = new Messagerie(new IdentificationDummy());  
  
        // On fait les tests comme on le souhaite dessus...  
    }  
}
```

Les mock objects

Les mocks objects sont plus complexes mais plus flexibles que les autres et c'est ceux là que l'on va privilégier pour l'activité intégrative en utilisant la librairie Mockito.

Contrairement aux autres, les mocks sont générés par une librairie, on a donc pas besoin de créer la classe nous même, il suffit juste de dire dans notre test que l'on souhaite créer un Mock et quelle valeur on veut que certaines méthodes retournent.

Ainsi les Mocks ont la simplicité des Fake objects mais sans avoir à créer la moindre classe soi-même.

Pour l'exemple précédent à la place de créer tout une classe on a simplement à définir ceci dans le test :

```
public class MessagerieTest {  
    // On demande à Mockito de créer un mock pour nous  
    @Mock private Identification identification;  
  
    @Test  
    void testLireMessages() {  
        // On configure le mock pour lui dire les paramètres et réponses attendues  
        when(identification.identify("toto", "mdp")).thenReturn(true);  
  
        // On l'injecte dans le constructeur de la messagerie  
        Messagerie messagerie = new Messagerie(identification);  
  
        // Enfin on peut faire les tests sur la messagerie comme on le souhaite...  
    }  
  
    // On peut ensuite réutiliser notre mock de la même façon pour d'autres tests  
}
```

Libraries

Il existe 2 librairies principales pour faire du *mocking* en Java, mais ici c'est Mockito qui a été privilégié.

Mockito

- facile d'utilisation
- configuration via annotation simple
- très grande communauté
- Choisi pour le cours

La documentation de Mockito est assez affreuse mais au moins elle est là, vous pouvez retrouver quelques liens intéressants [sur leur site](#), ainsi que [leur documentation officielle](#).

EasyMock

- Facile d'utilisation
- Configuration simple mais plus chiant que l'autre
- Moins utilisé que mockito

Spy objects

Les spy objects permettent de vérifier qu'une méthode a été appelée, de savoir combien de fois et avec quels arguments. Pour reprendre l'exemple précédent, si on imagine que la méthode `identify` est void, et ne retourne donc rien; on pourra tout de même la tester en vérifiant qu'elle a bien été appelée avec les bons arguments.

Cela peut être fait dans un Fake object mais est beaucoup plus compliqué à mettre en place, cela est en revanche trivial à faire avec Mockito :

```
public class MessagerieTest {
    @Mock private Identification identification;
    @Test
    void testLireMessages() {
        // On injecte la méthode dans le constructeur de la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests...

        // On peut ensuite par exemple aller vérifier que la méthode ~identify~ a été appelée exactement une fois
        // avec les paramètres "toto" et "mdp":
        verify(identification, times(1)).identify("toto", "mdp");
    }
}
```

De plus Mockito permet également d'espionner de vrais objets.

```
public class MessagerieTest {
    @Spy private Identification identification = new RealIdentification();

    @Test
    void testLireMessages() {
        // On injecte la classe espion dans la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests

        // On peut vérifier que l'identification a bien été appelée :
        verify(identification, times(1)).identify("toto", "mdp");

        // Note, si on le souhaite on pourrait même stub les méthodes de la vraie classe en faisant
        // when().thenReturn() par exemple
    }
}
```

```
}
```

Revision #1

Created 27 September 2023 11:32:16 by SnowCode

Updated 27 September 2023 11:34:32 by SnowCode