

Atelier 6 – partie 2 : les mocks

Table des matières

| | | |
|-------|---|---|
| 1 | Objectifs..... | 2 |
| 2 | Concepts..... | 2 |
| 2.1 | Les mocks | 2 |
| 2.1.1 | Généralités | 2 |
| 2.1.2 | Mockito..... | 5 |
| 2.2 | Gestion des dépendances d'une application Java..... | 6 |
| 2.2.1 | Généralités | 6 |
| 2.2.2 | Git et les dépendances | 6 |
| 3 | Exercices..... | 7 |
| 3.1 | Ajout des dépendances à un projet Maven | 7 |
| 3.2 | Les mocks via Mockito..... | 8 |
| 3.3 | Challenge optionnel..... | 9 |

1 Objectifs

| ID | Objectifs |
|------|--|
| AJ01 | Réaliser des vrais tests unitaires |
| AJ04 | Ajouter des dépendances à un projet Maven |
| AJ05 | Créer des stubs et des mocks à l'aide d'une librairie facilitant les tests |

2 Concepts

2.1 Les mocks

2.1.1 Généralités

Dans certains cas, les tests consistent à vérifier si l'on a bien appelé les bonnes méthodes avec les bonnes données. **Les mocks servent à tester le comportement d'un objet.**

Exemple : on veut faire persister des objets dans une base de données. Il faut vérifier que l'on envoie bien les bonnes données au DAO (Data Access Object, un objet permettant de regrouper les opérations vers une base de données).

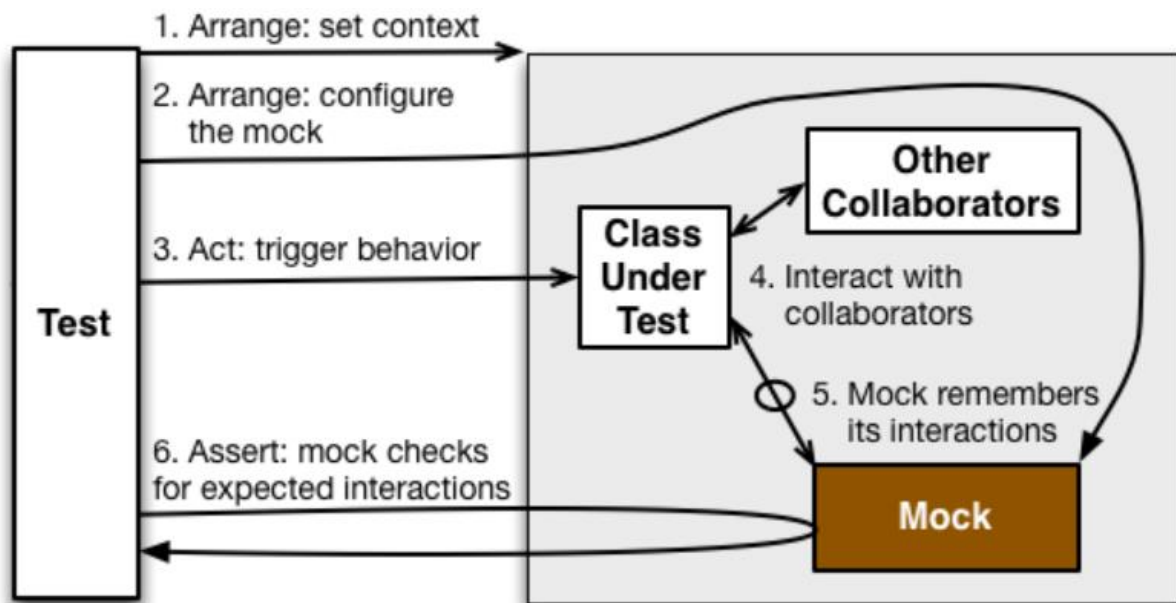
Dans ce cas le mock object vérifiera que l'on appelle bien les bonnes méthodes dans le bon ordre avec les bonnes données. Si l'on n'appelle pas la bonne méthode au bon moment, le mock object déclarera le test échoué.

A la fin il faut encore vérifier que l'on a bien appelé toutes les méthodes nécessaires (on pourrait ne rien avoir fait). C'est pourquoi l'on a une méthode de vérification finale qui déclarera le test échoué si au moins une des méthode devant être appelée ne l'a pas été.

Voici le cycle de vie d'un test avec des mocks :

1. **Arrange – Set context** : préparez l'objet sous tests ;
2. **Arrange – Configure the mock** : préparez les expectations (les attentes) dans le mock qui est utilisé par l'objet sous tests, principalement indiquez au mock quoi attendre en terme d'appels de méthodes et leurs arguments ;
3. **Act** : testez un comportement de l'objet sous tests, appelez la méthode à tester ;
4. **Interact** : l'objet sous tests fait son job en collaborant avec d'autres objets ;
5. **Mock remembers its interaction** : le mock retient ses interactions, les appels de méthodes, les arguments associés, ... ;
6. **Assert expectations** : vérifiez que les méthodes correctes ont été appelées dans le mock (avec éventuellement les données correctes en arguments) ; vous pouvez toujours utiliser des asserts pour vérifier l'état de l'objet sous tests, pour vérifier que la méthode testée a les effets désirés.

Ce cycle de vie peut être représenté ainsi :



[Fake and Mock Objects in Pictures, Bill Wake, <https://www.industriallogic.com/blog/mock-objects-pictures/>]

Il est à noter que, contrairement au cas des stubs, la vérification ne se fait pas dans la méthode de test mais bien dans le mock objet!

L'écriture d'objets de type mock peut s'avérer longue et fastidieuse, il s'agit de configurer chaque objet mocké pour qu'il réagisse comme souhaité. Cette configuration concerne :

- les méthodes invoquées : paramètres d'appel et valeurs de retour ;
- l'ordre d'invocation de ces méthodes ;
- le nombre d'invocations de ces méthodes.

Des bibliothèques ont donc été conçues pour rendre la création de ces objets fiable et rapide ; elles permettent de créer dynamiquement des objets généralement à partir d'interfaces ou de classes. Elles proposent fréquemment des fonctionnalités très utiles au-delà de la simple simulation d'une valeur de retour comme :

- la simulation de cas d'erreurs en levant des exceptions ;
- la validation des appels de méthodes ;
- la validation de l'ordre de ces appels ;
- la validation des appels avec un timeout.

Reprenons l'exemple de la fiche précédente avec la classe **Calculator** et le cycle de vie d'un test donné ci-dessus. Nous allons utiliser Mockito pour créer un stub pour la classe **DataService**. Nous ne devons donc plus créer de stub personnalisé et l'ancienne classe **DataServiceStub** va donc être remplacée par un Mock object.

La classe **Calculator** reste la même. Pour rappel, voici son contenu :

```
public class Calculator {  
    private DataService dataService;
```

```

public Calculator(DataService dataService) {
    this.dataService = dataService;
}

public int addToX(int a, int b) {
    int x = dataService.getData();
    return a + b + x;
}
}

public interface DataService {
    int getData();
}

```

Voici le code mis à jour pour **CalculatorTest** en utilisant la librairie Mockito :

```

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mockito;

public class CalculatorTest {

    private DataService dataService; // Le mock

    private Calculator calculator;

    @Before
    public void setUp() {
        // Arrange and Configure the Mock
        dataService = Mockito.mock(DataService.class); // Crée un mock de
        DataService
        calculator = new Calculator(dataService);
    }

    @Test
    public void testAddToX() {
        // Arrange
        int a = 3;
        int b = 4;
        int x = 5; // Valeur simulée pour getData
        Mockito.when(dataService.getData())
            .thenReturn(x); // Configuration du mock pour renvoyer la valeur x

        // Act
        int result = calculator.addToX(a, b);

        // Interact with Collaborators & Mock Remembers its Interactions

        Mockito.verify(dataService).getData(); // Vérifie que la méthode
        getData du mock a été appelée

        // Assert
        assertEquals(12, result); // Vérifie que le résultat est correct
    }
}

```

Dans cet exemple :

1. **Arrange** : on prépare le contexte du test, la préparation de l'objet à tester (**calculator**), et des paramètres qui seront donnés à la méthode testée **addToX**.
2. **Configure the Mock**: Dans la méthode **setUp**, nous utilisons la méthode **Mockito.mock(DataService.class)** pour créer le mock, que nous stockons dans la variable **dataService**. Ensuite, nous configurons le comportement du mock en utilisant **Mockito.when** pour qu'il renvoie une valeur simulée lors de l'appel à **getData**.
3. **Act**: Dans notre test **testAddToX**, nous appelons la méthode **addToX** de **Calculator** avec les valeurs **a** et **b**.
4. **Interact with Collaborators**: Nous utilisons **Mockito.verify** pour vérifier que la méthode **getData** du mock a bien été appelée.
5. **Mock Remembers its Interactions**: Cette étape est vide car Mockito se souvient automatiquement des interactions avec les mocks.
6. **Assert**: Enfin, nous utilisons une assertion pour vérifier que le résultat de la méthode **addToX** correspond à nos attentes.

Cela illustre comment on peut tester la classe **Calculator** en utilisant Mockito tout en suivant le cycle de vie d'un test avec des Mock.

2.1.2 Mockito

Dans ce cours, nous utiliserons la librairie **Mockito** pour gérer nos mocks.

Voici quelques instructions qui vont servir :

- Création d'un mock objet pour l'interface Stage :

```
Stage stage = Mockito.mock(Stage.class);
```

- Configurer les valeurs renvoyées lors d'appel de méthode sur un mock (stage) :

```
Mockito.when(stage.enregistrerMoniteur(moniteur)).thenReturn(true);
```

>signifie que l'appel de la méthode **enregistrerMoniteur** (avec comme paramètre **moniteur**) sur le mock renverra **true**.

```
Mockito.when(stage.enregistrerMoniteur(moniteur)).thenReturn(true).thenReturn(false);
```

>signifie que l'appel de la méthode **enregistrerMoniteur** (avec comme paramètre **moniteur**) sur le mock renverra **true** la première fois puis **false**.

```
Mockito.when(stage.enregistrerMoniteur(null)).thenThrow(IllegalArgumentException.class);
```

>signifie que l'appel de la méthode **enregistrerMoniteur** avec un paramètre **null** lance ladite exception.

- Vérifier qu'une méthode a été appelée (avec le bon paramètre) ou non sur un mock (stage) :

```
Mockito.verify(stage).enregistrerMoniteur(moniteur);
```

> vérifie que la méthode **enregistrerMoniteur** a bien été invoquée, avec comme paramètre **moniteur**, une et une seule fois.

```
Mockito.verify(stage, Mockito.times(nb)).enregistrerMoniteur(moniteur);
```

> vérifie que la méthode **enregistrerMoniteur** a bien été invoquée, avec comme paramètre **moniteur**, exactement **nb** fois.

```
Mockito.verify(stage, Mockito.never()).enregistrerMoniteur(moniteur);
```

> vérifie que la méthode **enregistrerMoniteur**, avec comme paramètre **moniteur**, n'a pas été invoquée.

```
Mockito.verify(stage, Mockito.atLeastOnce()).enregistrerMoniteur(moniteur);
```

> vérifie que la méthode **enregistrerMoniteur** a bien été invoquée, avec comme paramètre **moniteur**, au moins une fois.

- D'autres possibilités à découvrir dans la doc <https://www.javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

2.2 Gestion des dépendances d'une application Java

2.2.1 Généralités

Lors de la création d'un projet sous IntelliJ, il est possible de choisir un gestionnaire qui va faciliter le **build** de nos applications parmi **Maven** ou **Gradle**.

La partie qui nous intéresse le plus dans ces outils qui facilitent le **build** de nos projets, c'est la **gestion des dépendances** de nos programmes.

Maven ou **Gradle**, permettent de gérer facilement les librairies associées à nos applications via :

- un ou plusieurs fichier(s) de configuration identifiant notamment les versions des librairies à installer ;
- des commandes (pour télécharger, mettre à jour, supprimer des librairies,...) ;
- et des repository distants contenant les librairies à télécharger.

Pour nos ateliers, nous avons décidé que **Maven** serait le gestionnaire de dépendances de nos applications. **Maven** permettra notamment de gérer les librairies utiles à notre programme tout comme le ferait **npm** pour les librairies du monde **Node.js**. **Maven** contient un repository centralisé qui contient plein de projets que des développeurs comme nous ont voulu y mettre.

Pour un projet **Maven**, les dépendances d'une application seront indiquées dans le fichier **pom.xml**.

Nous verrons dans les exercices comment utiliser IntelliJ pour télécharger les dépendances indiquées dans **pom.xml**.

2.2.2 Git et les dépendances

Veuillez bien noter que lorsque l'on gère des dépendances de façon moderne, à l'aide de **Maven** ou tout autre outil similaire, on ne met jamais sous configuration le code source des dépendances via Git. Pensez donc, si vous utilisez un web repo comme **GitHub** ou **GitLab**, à bien ajouter un fichier **.gitignore** dans votre repo pour ignorer le tracking de vos dépendances.

En effet, on ne devrait jamais mettre en configuration du code qui ne nous appartient pas, et qui, de plus, est déjà si bien géré au niveau des versions par un autre outil (Maven, Gradle...).

On mettra donc en configuration le ou les fichier(s) s'occupant de lister les dépendances à installer par notre gestionnaire (pom.xml si Maven, build.gradle si Gradle, package.json si npm...), mais pas les répertoires contenant le code des librairies. Sous IntelliJ, pour générer un bon fichier **.gitignore**, vous

pouvez utiliser le plugin **.gitignore**. Pour l'installer : **File -> Settings... -> Plugins**, cliquez sur **Marketplace**, puis **.gitignore**, **Install**.

Plus d'information sur le plugin **.gitignore** d'IntelliJ : <https://github.com/JetBrains/idea-gitignore>.

3 Exercices

3.1 Ajout des dépendances à un projet Maven

Veuillez créer un nouveau Projet **Maven** au sein d'IntelliJ nommé **AJ_atelier06_partie2**. Pour ce faire :

- Dans IntelliJ, cliquez sur **New Project (File → New → Project** si vous n'êtes pas dans la fenêtre de départ).
- Donnez le nom **AJ_atelier06_partie2** à votre projet dans **Name**.
- Choisissez l'endroit où vous voulez créer votre projet dans **Location**.
- Choisissez **Maven** (et non IntelliJ !) comme **Build system**.
- N'hésitez pas, dans **Advanced Settings**, à donner un **GroupId** comme **be.vinci** par exemple.
- Cliquez sur **Create**.

Nous allons reprendre les mêmes classes et interfaces que celles utilisées lors de la partie 1 de cet atelier 6 : copiez/collez les packages **domaine** et **utils** au sein de votre nouveau projet dans **/src/main/java**.

Pour créer les mocks objects dont nous avons besoin, nous allons utiliser le framework Mockito (<https://site.mockito.org/>).


Mockito va donc être intégré au projet via **Maven**, via le fichier **pom.xml** qui décrit le projet.

L'**artifact ID**, combiné au **group ID** est l'identifiant unique de notre projet. C'est grâce à lui qu'on peut différencier notre projet des autres.

AJOUTER LES DÉPENDANCES À NOTRE PROJET AU SEIN DE POM.XML

- Ouvrez le fichier **pom.xml** et placez-vous dans le fichier (quelque part après la première ligne !).
- Sélectionnez **Generate...** dans le menu **Code** (le raccourci **ALT + INSERT** fonctionne aussi)
- Choisissez **Add dependency...**
- Dans le champ de recherche, à côté de la loupe, entrez **jupiter** et sélectionnez **JUnit Jupiter** en version **5.10.0** (ou une version plus récente) dans la liste qui apparaît.
- Cliquez sur **Add**.

Refaites les étapes précédentes en entrant **mockito** dans le champ de recherche et en sélectionnant **mockito-junit-jupiter** (**org.mockito:mockito-junit-jupiter**) en version **5.4.0** (ou une version plus récente) dans la liste des choix.

Cliquez sur le symbole  apparu en haut à droite pour charger les changements dans **Maven** (ou raccourci clavier **CTRL + MAJ + O**).

Dans votre fichier **pom.xml** devrait maintenant se trouver ceci (en dessous de <version>...) qui indique les dépendances ajoutées :

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0</version>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>5.4.0</version>
  </dependency>
</dependencies>
```

Les dépendances apparaîtront également dans les **External Libraries** au sein de l'explorateur du projet.

3.2 Les mocks via Mockito

Les tests que nous allons maintenant effectuer portent sur les transitions d'états ; **l'état correspond toujours au nombre de stages associés à un moniteur !** Il faut tester si les méthodes interagissent correctement avec les autres objets et leur donnent les valeurs attendues. Il faut implémenter ces tests avec des mocks objects.

Nous avons déjà établi un plan de tests pour la méthode **ajouterStage (Stage stage)** :

| TC # | Etat | Méthode | Paramètres | Etat Suivant | Résultat attendu | Valeur retour | Exception | Objet participant | Méthode appelée | Paramètres |
|------|------|---------------|----------------------------|--------------|--|---------------|-----------|-------------------|--|-------------|
| 1 | 0 | ajouter Stage | stage valide | 1 | Le stage est ajouté | true | | stage | enregistrer Moniteur | le moniteur |
| 2 | 1 | ajouter Stage | stage valide semaine libre | 2 | Le stage est ajouté les autres sont toujours présents | true | | stage | enregistrer Moniteur | le moniteur |
| 3 | 2 | ajouter Stage | stage valide semaine libre | 3 | Le stage est ajouté les autres sont toujours présents | true | | stage | enregistrer Moniteur | le moniteur |
| 4 | 3 | ajouter Stage | stage valide semaine libre | 4 | Le stage est ajouté, les autres sont toujours présents | true | | stage | Enregistrer Moniteur | Le moniteur |
| 5 | 4 | ajouter Stage | stage déjà présent | 4 | stage non ajouté | false | | stage | enregistrer Moniteur appelé une seule fois | |
| 6 | 4 | ajouter Stage | semaine déjà prise | 4 | stage non ajouté | false | | stage | enregistrer Moniteur | |

| TC # | Etat | Méthode | Paramètres | Etat Suivant | Résultat attendu | Valeur retour | Exception | Objet participant | Méthode appelée | Paramètres |
|------|------|---------------|--|--------------|------------------|---------------|-----------|-------------------|-------------------------------------|------------|
| | | | | | | | | | jamais appelée | |
| 7 | 4 | ajouter Stage | stage possédant déjà un autre moniteur | 4 | stage non ajouté | false | | stage | enregistrer Moniteur jamais appelée | |
| 8 | 4 | ajouter Stage | stage non présent avec comme moniteur celui auquel on veut ajouter le stage (semaine libre, ...) | 5 | stage ajouté | true | | stage | enregistrer Moniteur jamais appelée | |
| 9 | 0 | ajouter Stage | stage sans moniteur pour un sport pour lequel le moniteur n'est pas compétent | 0 | stage non ajouté | false | | stage | enregistrer Moniteur jamais appelée | |

Comme nous devons vérifier que la méthode **ajouterStage** de la classe **MoniteurImpl** met bien en place les deux côtés de l'association en appelant la méthode **enregistrerMoniteur** de l'objet **stage**, nous allons devoir construire un (ou plusieurs) mock(s) object(s) pour la classe **Stage**. La raison principale d'un mock object est de vérifier que l'on appelle bien la ou les méthodes attendues avec le ou les bons paramètres.

ÉCRITURE DES TESTS

Écrivez maintenant les tests correspondant au plan de tests dans une classe JUnit **MoniteurImplTest** que vous générez comme expliqué à la partie 1 de cet atelier 6. Nommez vos méthodes de test en fonction des cas décrits dans le plan de tests. Par exemple : **TestMoniteurTC1**. Comme précédemment, vous pouvez écrire une méthode **private** s'appelant **amenerALEtat(int etat, Moniteur moniteur)** afin d'amener un objet **Moniteur** à l'état à tester (en utilisant **Mockito** et non les stubs de la partie précédente !).

Si vous avez besoin d'inspiration pour configurer vos mocks, n'hésitez pas à (re)lire le §2.1.2 Mockito.

3.3 Challenge optionnel

Faites un plan de tests pour la méthode **supprimerStage(Stage stage)** de la classe **MoniteurImpl** et complétez la classe de tests afin de le réaliser.