Java Avancé

Atelier 6 – partie 1 : les stubs

Table des matières

1 Objectifs

ID	Objectifs
AJ01	Réaliser des vrais tests unitaires
AJ02	Abstraire l'implémentation de classes par l'utilisation d'interfaces
AJ03	Créer ses propres stubs

2 Concepts

2.1 Réalisation de tests unitaires

On parle de vrais tests unitaires lorsque l'on teste qu'une seule classe, sans tester ses dépendances.

Par exemple, si l'on teste une classe qui a des attributs qui sont des instances d'autres classes, alors il est important de ne pas tester le code de ces autres classes. Nous parlerions de tests d'intégration si l'on testait le fonctionnement de plusieurs classes qui communiquent.

Pour faire de vrais tests unitaires, nous allons simuler le comportement des toutes les instances qui n'appartiennent pas au type de la classe testée.

Il existe différents types de faux objets, nous n'en n'examinerons que deux dans nos ateliers :

- les stubs ;
- les mocks objects.

NB : on utilise aussi, en fonction des besoins, des fake objects, ou des spies.

2.2 Abstraction de l'implémentation des classes

Lorsque nous souhaitons créer du code qui peut être facilement testable de façon unitaire, nous allons abstraire l'implémentation de classes par l'utilisation d'interfaces ; ainsi, lors d'interactions entre des objets de différents types, on ne doit pas se soucier de l'implémentation des classes ; l'auteur d'une classe peut mettre à jour son implémentation sans casser les interactions avec celle-ci ; cela participe à la mise en place d'un couplage faible entre objets.

Un faible couplage entre objets facilite grandement les tests unitaires car nous allons pouvoir facilement créer des faux objets pour tous les types qui ne correspondent pas à la classe sous tests.

Ce concept va être clarifié via les prochains exercices.

2.3 Les stubs

Les stubs sont des **objets sans aucune intelligence renvoyant systématiquement la même réponse** à l'appelant et n'effectuant aucun traitement. **Les stubs servent à tester l'état d'un objet**.

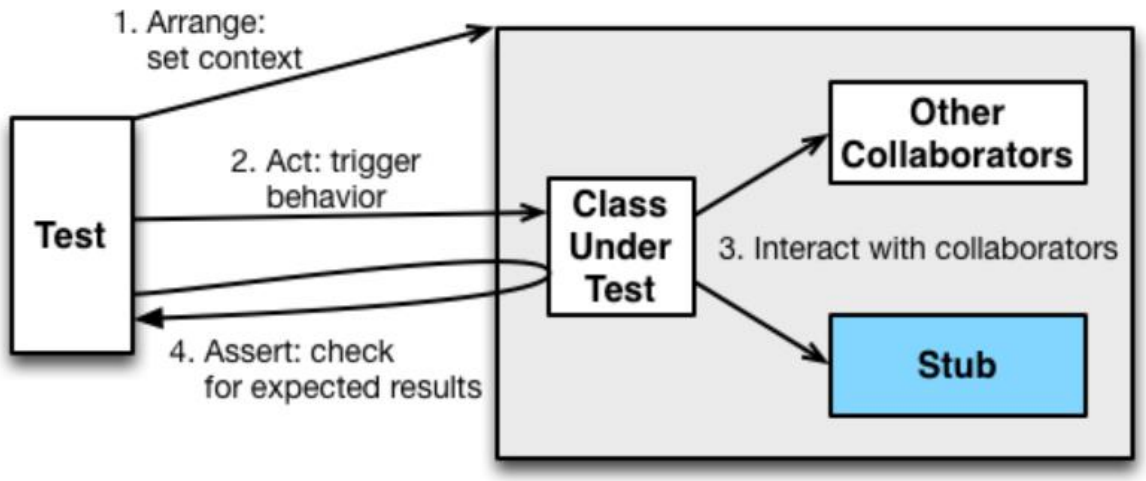
Le stub est utile lorsque, pour effectuer un test, on a besoin des réponses données par ces objets mais que le test ne porte pas sur notre interaction avec ces objets.

Exemple : on doit traiter des objets provenant d'une base de données. Le test ne porte pas sur notre interface avec la base de données mais sur le traitement des objets. Les stubs nous renverront des objets à traiter.

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

1. **Arrange** – **Set context** : préparez l'objet sous tests, ses collaborateurs « stubs » et les connecter;
2. **Act** : testez un comportement de l'objet sous tests, appelez la méthode à tester ;
3. **Interact** : l'objet sous tests fait son job en collaborant avec d'autres objets ;
4. **Assert** : utilisez des asserts pour vérifier l'état de l'objet, pour vérifier que la méthode testée a les effets désirés.

Ce cycle de vie peut être représenté ainsi (modèle AAA) :



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

Supposons que vous ayez une classe **`Calculator`** que vous souhaitez tester. Cette classe a une méthode **`addToX`** qui additionne deux nombres à une donnée externe (x). Vous voulez tester cette méthode en utilisant un stub pour simuler la dépendance à une source de données externe.

```
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();
}
```

Vous pourriez créer un stub en implémentant votre propre classe qui implémente l'interface **`DataService`**.

Tout d'abord, on créerait une classe **`DataServiceStub`** qui implémente l'interface **`DataService`** :

```
public class DataServiceStub implements DataService {
  private int data;

  public DataServiceStub(int data) {
    this.data = data;
  }

  @Override
  public int getData() {
    return data;
  }
}
```

Voici un exemple de test unitaire en utilisant JUnit et le cycle de vie donné à la page précédente :

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {

  private Calculator calculator;
  private DataServiceStub dataServiceStub;

  @Before
  public void setUp() {
    // Arrange
    dataServiceStub = new DataServiceStub(5); // Crée un stub personnalisé
avec une valeur de 5

    calculator = new Calculator(dataServiceStub);
  }

  @Test
  public void testAddtoX() {
    // Arrange (optionnel, car déjà fait dans le setUp)

    // Act
    int result = calculator.addToX(3, 4);

    // Interact (pas besoin de vérifier l'interaction car c'est un stub
personnalisé)

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

Dans cet exemple :

- La section "Arrange" consiste à configurer l'environnement de test. Nous créons un stub pour **`DataService`** à l'aide de notre classe **`DataServiceStub`** et définissons son comportement à renvoyer la valeur 5 lorsqu'il est appelé.

- Dans la section "Act", nous appelons la méthode **`addToX`** de la classe **`Calculator`** que nous testons.
- Dans la section "Interact", il n'y a rien à faire car nous utilisons un stub personnalisé (nous savons que la fonction **getData** du stub sera appelée).
- Enfin, dans la section "Assert", nous vérifions que le résultat de la méthode **`addToX`** est conforme à nos attentes à l'aide d'une assertion.

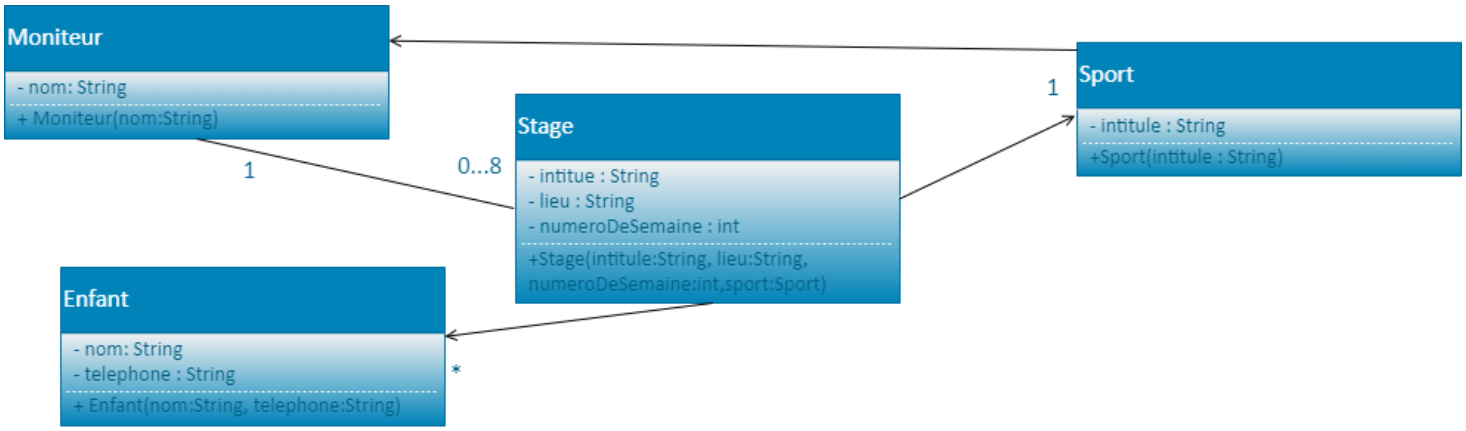
Cela illustre comment structurer un test unitaire en Java avec un stub en utilisant le modèle AAA pour tester la classe **`Calculator`**.

3 Exercices

3.1 Introduction

Une ASBL organise des stages sportifs pour les enfants pendant les vacances d'été. Afin de faciliter l'organisation, elle désire se munir d'un logiciel de gestion de ces stages.

L'analyse a déterminé les classes **`Stage`**, **`Moniteur`**, **`Sport`**, **`Enfant`** :



En plus des champs liés aux associations, on a que :

- Un stage stocke un intitulé ("stage d'initiation à la natation"), un lieu ("piscine du Blocry à LLN") et un numéro de semaine (les semaines de vacances d'été ont été numérotées de 1 à 8).
- Un moniteur stocke son nom.
- Un sport stocke son intitulé.
- Un enfant stocke son nom et son numéro de téléphone.

Une implémentation du domaine est fournie. Elle est commentée en javadoc.

Votre travail aujourd'hui consiste à tester la classe **`Moniteur`**. Pour des raisons de temps, nous testons uniquement l'association entre la classe **`Moniteur`** et la classe **`Stage`** dans le sens **`Moniteur`** vers **`Stage`**.

Veuillez créer un nouveau Projet **Maven** au sein d'IntelliJ nommé **AJ_atelier06_partie1**. 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_partie1** 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**.

Récupérez l'archive qui se trouve sur Moodle (archive-for-students.zip) et qui contient les classes du domaine. Copiez/collez les répertoires **domaine** et **utils** au sein de votre nouveau projet dans **/src/main/java**.

3.2 Abstraction de l'implémentation des classes

Dans un premier temps, il est demandé de modifier le code afin de limiter le couplage entre les objets. Il s'agit de rendre le code indépendant et maintenable en ne dépendant pas de choix d'implémentation limitatif. Il faut donc extraire les interfaces de chaque classe du domaine.

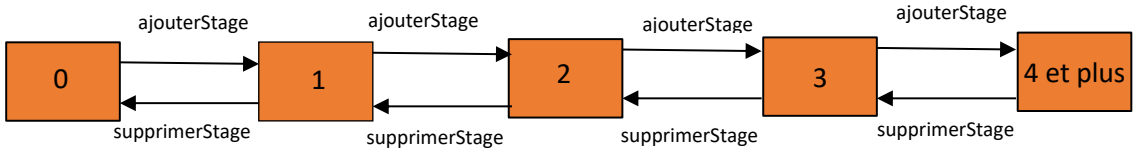
Par convention, il est demandé que les interfaces conservent le nom métier : **Stage**, **Sport**, etc. Les classes qui implémentent celles-ci portent le nom de l'interface suivi de **`Impl`**.

Pour extraire une interface d'une classe avec IntelliJ, veuillez suivre les étapes ci-dessous :

- faites un clic droit sur le nom de la classe et choisissez **Refactor → Extract Interface …** ;
- choisissez « **Rename original class and use interface where possible** »[1] ;
- entrez le bon nom pour la classe(nom de l'interface suivi de **Impl**) dans le champ « **Rename implementation class to :** » ;
- sélectionnez toutes les méthodes afin qu'elles soient mises dans l'interface ;
- sélectionnez **copy** pour que la javadoc soit copiée dans l'interface tout en restant dans la classe ;
- cliquez sur **Refactor**.

3.3 Les stubs

Les analystes ont mis en place un diagramme des différents états du **`Moniteur`**. Ce diagramme est relativement simplifié par le fait que l'**état** correspond au **nombre de stages** de celui-ci.



Par exemple, après avoir appelé deux fois, avec succès, la méthode **`ajouterStage`** sur un moniteur, celui-ci aura donc 2 stages : l'état de ce moniteur sera donc de 2 !

Ce diagramme aurait pu poursuivre ses transitions à l'infini mais cela n'aurait aucun intérêt puisqu'après quelques stages le comportement est toujours identique.

Sur base de ce diagramme, le plan de tests (uniquement pour les méthodes **ajouterStage** et **supprimerStage**) suivant a été mis au point :

TC #	Etat	Méthode	Paramètres	Etat Suivant	Résultat attendu	Valeur retour	Exception
1	0	`ajouterStage`	stage valide (sport dans lequel le moniteur est compétent, stage sans moniteur)	1	Le stage est ajouté	true	
2	1	`ajouterStage`	stage valide semaine libre	2	Le stage est ajouté l'autre est toujours présent	true	

[1] Cela a pour effet que l'interface va conserver le nom métier (celui de la classe) et que vous pourrez choisir le nouveau nom pour la classe. De plus, ce sera l'interface qui sera utilisée dans les autres classes quand c'est possible.

TC #	Etat	Méthode	Paramètres	Etat Suivant	Résultat attendu	Valeur retour	Exception
3	2	`ajouterStage`	stage valide semaine libre	3	Le stage est ajouté les autres sont toujours présents	true	
4	3	`ajouterStage`	stage valide semaine libre	4	Le stage est ajouté les autres sont toujours présents	true	
5	4	`supprimerStage`	déjà présent	3	Le stage n'est plus présent, les autres sont toujours présents	true	
6	3	`supprimerStage`	déjà présent	2	Le stage n'est plus présent, les autres sont toujours présents	true	
7	2	`supprimerStage`	déjà présent	1	Le stage n'est plus présent, les autres sont toujours présents	true	
8	1	`supprimerStage`	déjà présent	0	Il n'y a plus aucun stage	true	
9	4	`ajouterStage`	stage déjà présent	4	stage non ajouté	false	
10	4	`ajouterStage`	semaine déjà prise	4	stage non ajouté	false	
11	4	`supprimerStage`	stage non présent	4	Tous les stages sont présents rien n'a été supprimé	false	
12	4	`ajouterStage`	stage possède déjà un autre moniteur	4	stage non ajouté	false	
13	4	`ajouterStage`	stage non présent avec comme moniteur celui auquel on veut ajouter le stage (semaine libre, …)	5	stage ajouté	true	
14	4	`ajouterStage`	stage sans moniteur pour un sport pour lequel le moniteur n'est pas compétent	4	stage non ajouté	false	

Afin d'implémenter ces tests de façon vraiment unitaire, on va utiliser des stubs. Dans un projet **Maven**, les tests se trouveront d'office dans le répertoire **/src/test/java**.

Créez une classe de tests JUnit **MoniteurImplTest** pour la classe **MoniteurImpl** de cette façon-ci :

1. Ouvrir la classe **MoniteurImpl** que vous voulez tester
2. Se positionner sur le nom de la classe au sein du code, faire un clic droit et choisir « **Show Context Action** »
3. Choisir « **Create Test** »
4. Dans Testing Library, il faut choisir « **JUnit5** ». SI le message « **JUnit5 library not found in the module** » apparaît, cliquez sur **Fix** et ensuite sur **OK**.
5. Vérifiez que le package de destination indiqué est bien le même que le package de la classe à tester (package **`domaine`**).

6. Dans **Generate**, sélectionnez **setUp/@Before**.
7. Sélectionnez ensuite les méthodes que vous voulez tester (**ajouterStage** et **supprimerStage**) et cliquez sur **OK**. Cela devrait générer votre classe de test dans **/src/test/java/domaine/MoniteurImplTest**.

OBSERVATIONS & QUESTIONS

Commencez par jeter un œil au code de la première méthode que nous souhaitons tester de façon unitaire : **ajouterStage** de **MoniteurImpl**. Cette méthode reçoit un objet de type **Stage**.

Comme cet objet ne fait pas partie du type de la classe testée, nous découvrons que nous allons devoir prochainement créer un stub pour l'interface **`Stage`**.

De plus, dans le code de **ajouterStage** de **MoniteurImpl**, on voit que nous allons devoir créer un autre stub. Vous l'avez découvert ?

Voici le code qui répondrait à la question :

```
if (!stage.getSport().contientMoniteur(this))
```

Comme pour pouvoir ajouter un stage à un moniteur, il faut pouvoir vérifier qu'il est compétent dans le sport du stage et que c'est l'interface **`Sport`** qui contient une méthode (méthode **`contientMoniteur`**) permettant de savoir si un moniteur est compétent dans un sport ou non, il faut aussi créer un stub pour l'interface **`Sport`**. Ce stub vous est déjà fourni à titre d'exemple : récupérez la classe **`SportStub`** et placez-la dans le package **domaine** de votre module de test. Vous pouvez constater que la classe **`SportStub`** contient un unique attribut (**`contientMoniteur`**) qui sera initialisé dans le constructeur et que la méthode **`contientMoniteur`** renvoie la valeur de cet attribut. Ainsi, on pourra, en fonction de l'initialisation, créer un sport dans lequel le moniteur est compétent (dans le cas où **`contientMoniteur`** vaut **`true`**) ou non (dans le cas où **`contientMoniteur`** vaut **`false`**).

Vous devez maintenant écrire la stub **`StageStub`** pour l'interface **`Stage`**. Avant d'écrire le stub il faut déterminer les méthodes pour lesquelles il faut pouvoir configurer le retour. Cela se fait en fonction des tests qu'on désire effectuer. On veut pouvoir créer un stage pour un sport dans lequel le moniteur est compétent ou non, …

Avec IntelliJ, générez le stub **`StageStub`** pour l'interface **`Stage`** (une simple classe implémentant **`Stage`** et définissant déjà toutes ses méthodes) et veillez à ce qu'elle se situe bien dans le package **`domaine`** du module de tests[2]. De manière analogue à ce qui a été fait dans **`SportStub`**, vous devrez aussi définir des attributs afin de pouvoir configurer les valeurs de retours des méthodes de la classe **`Stage`** utilisées pour les tests. Principalement en regardant le code à tester, **veuillez lire attentivement le code de la méthode `ajouterStage` de `MoniteurImpl`**, les appels de méthodes sur l'objet de type **`Stage`**, et aussi à l'aide du plan de tests, on constate qu'il faudra

[2] Pour générer une classe implémentant une interface avec IntelliJ, il faut ouvrir l'interface, se positionner sur le nom de l'interface dans le code, faire un clic droit, choisir « **Show Context Action** » et ensuite choisir « **Implement Interface** ». Dans la fenêtre qui s'ouvre, entrez le nom de la classe et choisissez le bon package et le bon répertoire de destination.

pouvoir configurer les valeurs de retour des méthodes **`getNumeroDeSemaine`**, **`getSport`** et **`getMoniteur`.**

Écrivez ensuite les tests correspondant au plan de tests dans la classe JUnit **`MoniteurImplTest`**. Nommez vos méthodes de test en fonction des cas décrits dans le plan de tests. Par exemple : **`TestMoniteurTC1`.**

Vous pouvez écrire une méthode **`private`** s'appelant **`amenerALEtat(int etat, Moniteur moniteur)`** afin d'amener un objet **`Moniteur`** à l'état à tester (ce qui veut dire lui ajouter des stages afin d'arriver au nombre désiré pour le test.). **Par exemple, si l'on veut amener un moniteur à l'état 2, il faudra que la méthode `amenerALEtat` fasse appelle deux fois à la méthode `ajouterStage` sur un moniteur !** Faites attention au fait qu'un moniteur ne peut pas avoir deux stages la même semaine.

N'oubliez pas que vous devez vérifier que le nouvel état (pour rappel, le nombre de stages) est atteint et que le résultat est bien celui attendu (valeur de retour et effets de l'appel).