Ateliers « Mise en pratique des tests avec JUnit »

Pré-requis :

- JDK 8+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)
- Librairie lombok : https://projectlombok.org/downloads/lombok.jar

Support et énoncé des ateliers disponibles à : https://github.com/dthibau/junit

Solutions des ateliers disponibles à : https://github.com/dthibau/junit-solutions

TP1 : Concepts JUnit

Objectifs

L'objectif de ce TP est de se familiariser avec les concepts de JUnit. En particulier, les notions de TestCase, de Fixture. Ensuite, l'approche XP est entrevue.

1.1 : Mise en place du projet

Créer un projet Maven et ajouter la dépendance suivante dans le *pom.xml*

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>
```

Récupérer la classe ***org.formation.Money***

1.2 : Premières méthodes de test

Nous disposons d'une classe *Money* qui comprend deux attributs : le montant (amount) et la monnaie (currency). Cette classe contient déjà une méthode permettant d'additionner deux objets de type *Money*.
Nous proposons d'écrire les TestCase permettant de tester la méthode *add()* et la méthode *equals()*

1. Créer la classe *MoneyTest*.
2. Implémenter les méthode vous paraissant adéquates pour tester les différents cas de test

1.3 : Approche TDD

Jusqu'à maintenant, nous avions utilisé des sommes d'argent de même type.

Pour additionner des sommes d'argent de monnaie différente, nous créons un nouvelle classe *MoneyBag* qui représente une collection de sommes d'argent de monnaies différentes.

Les classes ***Money*** et ***MoneyBag*** implémentent l'interface ***IMoney*** qui spécifie une seule méthode :

IMoney add(IMoney money) .

Avant d'implémenter la méthode pour ces 2 classes, définir les différents cas de test et les implémenter dans une classe de test. Réfléchir à la notion de *MoneyBag* vide

Procéder comme suit :

1. Définir les fixtures nécessaires et les initialiser
2. Écrire les méthodes de tests et s'assurer qu'elles échouent
3. Implémenter la méthode *add()* de la classe ***Money***, réexécuter les tests
4. Implémenter la méthode *add()* de la classe ***MoneyBag*** et s'assurer que tous les tests passent

1.4 : Exécution Maven

Quelle plugin et quelle version par défaut correspond à la phase maven ***test*** ?

Modifier le ***pom.xml*** afin que les tests JUnit5 soit détectés.

TP2 : Matcher *Hamcrest*

2.1: Ajout de dépendance

Ajouter la dépendance Hamcrest dans le *pom.xml*

```
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>
```

2.2: Matchers

Réécrire toutes les assertions avec des Matchers. Vous pouvez utiliser :

- Des contraintes sur le type
- Des contraintes sur des propriétés de JavaBean
- Des contraintes sur les collections
- Des *FeatureMatcher*
- Des opérateurs de coimbinaison

Faire échouer des tests pour visualiser le reporting

TP3 : M*ockito*

3.1: Ajout des dépendances

Ajouter les dépendances mockito-core et les extensions JUnit5

```
<dependency>
   <groupId>org.mockito</groupId>
```

```
    <artifactId>mockito-core</artifactId>
    <version>3.6.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.6.0</version>
    <scope>test</scope>
</dependency>
```

3.2: Configuration de Mock

Récupérer l'interface ***org.formation.ConvertService*** fournie

En utilisant l'interface *ConvertService*, écrire les tests unitaires et l'implémentation de la méthode de *MoneyBag*
public Money convertInto(String destinationCurrency)

Cette méthode convertit toutes les monnaies d'un porte monnaie dans une devise donnée.

3.3: Vérifications

Utiliser la méthode ***verify()*** pour vérifier les appels à *ConvertService* (nombre et arguments)

TP*4 : Compléments, tests paramétrés*

4.1: Tests paramétrés

Réécrire les tests de la méthode ***convertInto*** de ***MoneyBag*** sous forme de tests paramétrés

4.2 ParameterResolver

Reprendre la classe ***org.formation.validator.CurrencyValidator***

Écrire une méthode de test prenant en paramètre une String

Injecter plusieurs valeurs invalides dans la méthode via l'annotation *@RepeatedTest* et l'extension *ParameterResolver*

Effectuer les bonnes assertions

TP5 *: DBUnit*

Créer un projet Maven avec les dépendances sur h2 et DBUnit :

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>2.7.0</version>
```

```
  <scope>test</scope>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.200</version>
  <scope>test</scope>
</dependency>
```

- Visualisez le script de création de schéma fourni : ***schema.sql***
- Mettre en place un jeu de données remplissant les tables fournisseur et produit
- Mettre en place une première méthode de test qui vérifie qu'à l'exécution la table fournisseur est bien identique à celle du dataset original.
- Écrire une méthode de test qui charge un DataSet attendu, insère un nouveau fournisseur en appelant la classe fournie ***FournisseurDao*** et vérifie que le dataset réel est bien identique au dataset attendu

TP6 *: SpringBoot*

6.1: Mise en place

Importer le projet Maven fourni. Ce projet utilise la librairie Lombok. Pour metter en place lombok dans Eclipse :

- Mettre lombok.jar dans le répertoire de STS
- Ajouter dans SpringToolSuite4.ini la ligne :
  -javaagent:lombok.jar

Le projetr est une application web développée avec SpringBoot, offrant 3 principaux point d'accès :

- http://localhost:8080 : Application web
- http://localhost:8080/swagger-ui.html : Documentation REST API
- http://localhost:8080/actuator : Urls de monitoring

L'application est sécurisée (Sécurité stateful pour l'application Web ou stateless/JWT pour l'API Rest). Les utilisateurs sont définis dans le fichiers ***src/main/resources/users.csv***

L'application utilise SpringData pour accéder à une base de données relationnelles.

L'implémentation utilisée est une base embarquée h2. Dans ce cas, *SpringBoot* configure automatiquement *Hibernate* afin qu'il crée et initialise la base à chaque démarrage de l'application.

6.2 Tests auto-configurés

A chaque mise en place des tests, il est conseillé de se faire injecter le contexte applicatif Spring

(ApplicationContext) afin de pouvoir inspecter les beans configurés

@DataJpaTest

Ecrire une classe de test vérifiant le bon fonctionnement de la méthode *Optional<Fournisseur> findByReference(String reference);*

@JsonTest

Ecrire une classe de test vérifiant le bon fonctionnement de la sérialisation/désérialisation de la classe Produit

@WebMVCTest

Utiliser *@WebMVCTest* pour tester une méthode de *ProduitRestController* en utilisant un *mockMVC*

Utiliser les annotations @WithMockUser pour contourner la sécurité

TP7 *: Maven et CI*

7.1 Distinction unitaires et intégration

Sur le projet précédent, mettre en place une configuration Maven afin que les tests de la couche *controller* soit effectué dans la phase d'intégration.

7.2 Couverture des tests

Mettre en place JaCoCo et visualiser les rapports de couverture de test