AVIGNON UNIVERSITÉ

Rapport TP3
Pokéunit

Étudiant
Michel Marie LAMAH

14 avril 2023

Master Informatique
Intelligence Artificielle

UE Génie logiciel avancé
ECUE Techniques de test

Responsables
Daniel SALAS
Emmanuel FERREIRA

UFR
SCIENCES
TECHNOLOGIES
SANTÉ

AVIGNON UNIVERSITÉ
CERI

CENTRE
D'**ENSEIGNEMENT**
ET DE **RECHERCHE**
EN **INFORMATIQUE**
ceri.univ-avignon.fr

Sommaire

1 Introduction

Dans cet TP[1], il sera question de mettre en pratique les notions de tests unitaires et de doublures vu en classe.

Pour réaliser ce travail, j'utiliserai :

- **IDE** : Intellij IDEA [1]
- **Dépôt de repository Git** : Github [2]

Toutes les modifications réalisées ici se trouvent sur la branche **TP03** de mon repository [3].

Mais si vous souhaitez voir toutes les modifications effectuées depuis le TP1 jusqu'à maintenant vous pouvez voir la branche master [2].

2 Configuration du répertoire de tests

Pour pouvoir réaliser les tests unitaires, la premières des choses pour nous sera de configurer le projet en lui spécifiant le répertoire des fichiers de tests.

Et celà se fait facilement à travers le fichiers pom.xml. Heureusement pour nous lors du premier tp 1, le répertoire de tests avait été créé automatiquement par Intellij lorsque nous avons transformé notre projet en un projet Maven.

Toutefois cela ne nous empêche pas quand même de voire si maven l'a bien configuré. Pour cela, on exécute la commande ci-dessous :

```
mvn help:effective-pom
```

Console 1. Voire le fichier pom.xml de notre projet

Et il suffit de recherche alors la balise <build />, comme indiqué ci-dessous :

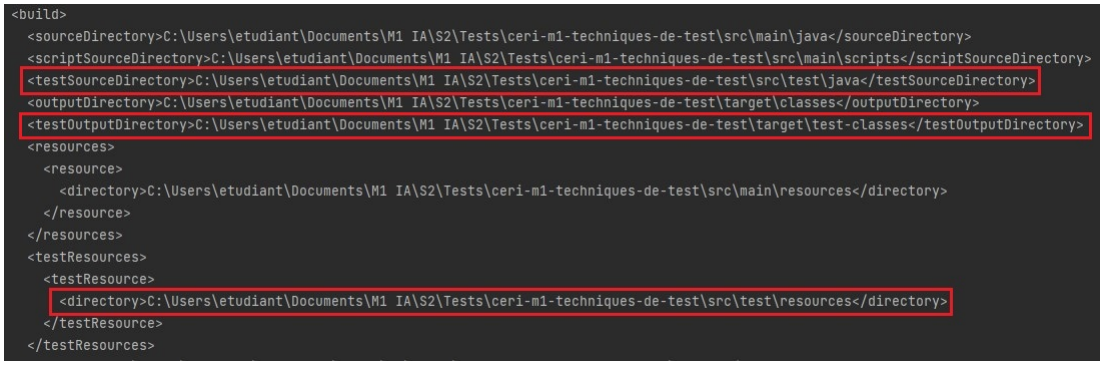

Figure 1. Répertoire de tests

Donc si ce n'était pas configuré, il fallait juste ajouter le bout de code indiqué ci-dessous :

```
<build>
    <testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>
    <testOutputDirectory>${project.basedir}/target/test-classes</testOutputDirectory>
    <testResources>
        <testResource>
            <directory>${project.basedir}/src/test/resources</directory>
        </testResource>
    </testResources>
</build>
```

Listing 1. Répertoire de test

1. https://www.jetbrains.com/fr-fr/idea/
2. https://github.com/

Remarque : `$project.basedir` est une propriété implicite des fichiers pom.xml qui représente la racine du projet

Après les modifications dans Listing 1 et en retapant la commande Console 1 on obtiendra les mêmes résultats que dans l'image Figure 1.

Il ne nous reste qu'à créer maintenant le package **`fr.univavignon.pokedex.api`** et pour celà, on effectue un clique droit sur le répertoire **`src\test\java`** puis :

New → **Package** → fr.univavignon.pokedex.api

On obtient alors la structure ci-dessous :

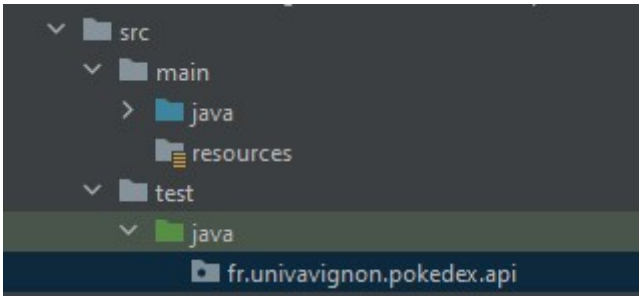

Figure 2. Structure du projet

3 Les tests unitaires

Une fois le répertoire de test spécifié, il va nous falloir maintenant créer nos tests, ci-dessous la liste des tests à créer :

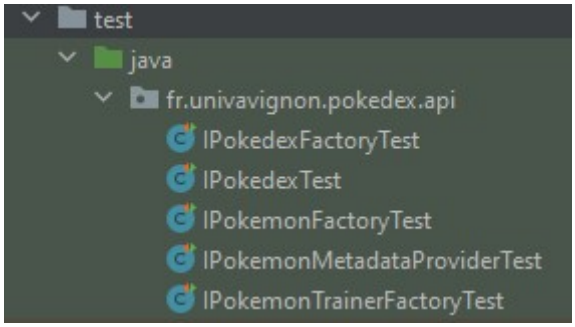

Figure 3. Les classes de tests

Dans la l'image Figure 3 ci-dessous, chaque cas de test **[nom_interface]Test** permettra de tester l'interface **[nom_interface]**

3.0.1 Le usecase IPokedexFactoryTest

Le usecase IPokedexFactoryTest permet de tester l'interface IPokedexFactory et cette interface ne contient qu'une seule méthode du coup on aura qu'un seul test unitaire pour cette classe.

Ci dessous la signature de la méthode que contient l'interface IPokedexFactory

```
IPokedex createPokedex(IPokemonMetadataProvider metadataProvider, IPokemonFactory
    pokemonFactory)
```

La méthode **createPodex** prend en paramètre une instance de l'interface IPokemonMetadataProvider et l'interface IPokemonFactory puis retourne une instance de l'interface IPokedex.

Vu que nous n'avons pas d'abord l'implémentation de ces interfaces, nous utilisons alors les mocks pour créer des doublures qui jouerons leurs rôles.

Pour réaliser tout celà, j'ai donc créé deux methodes :

- **start** : qui est une fixture qui se charge de :
  - créer le mock pour l'interface IPokedexFactory
  - créer le mock pour l'interface IPokedex
  - créer le mock pour l'interface IPokemonMetadataProvider
  - créer le mock pour l'interface IPokemonFactory
- **testCreatePokedex** : qui va permettre de tester le cas createPokedex et vu que cette méthode prennait en paramètre un IPokemonMetadataProvider et un IPokemonFactory alors nous utilisons les mocks précédement créés pour le tester.

3.0.2 Le usecase IPokemonFactoryTest

Le usecase IPokemonFactoryTest permet de tester l'interface IPokemonFactory et cette interface ne contient qu'une seule méthode du coup on aura qu'un seul test unitaire pour cette classe.

Ci dessous la signature de la méthode que contient l'interface IPokemonFactoryTest

```
Pokemon createPokemon(int index, int cp, int hp, int dust, int candy)
```

La méthode **createPokemon** prend en paramètres les caractéristiques d'un Pokemon puis crée ce Pokemon et le retourne.

De même que précédemment vu que nous n'avons pas son implémentation, nous allons alors utiliser les mocks pour réaliser cette tâche.

Pour cela je crée aussi deux méthodes :

- **start** : qui est une fixture qui se charge de :
  - créer le mock pour l'interface IPokemonFactary et y affecter un comportement
  - créer créer quelques instances de Pokemon
- **testCreatePokemon** : qui va utiliser les mocks précédemment créés pour tester la méthode createPokemon

3.0.3 Le usecase IPokemonMetadataProviderTest

Le usecase IPokemonMetadataProviderTest permet de tester l'interface IPokemonMetadataProvider et cette interface ne contient qu'une seule méthode du coup on aura qu'un seul test unitaire pour cette classe.

Ci dessous la signature de la méthode que contient l'interface IPokemonMetadataProvider

```
PokemonMetadata getPokemonMetadata(int index) throws PokedexException
```

Cette méthode prend en paramètre un index et retourne le Données du Pokemon correspondant s'il existe dans le cas contraire lève une exception.

De même que précédemment vu que nous n'avons pas son implémentation, nous allons alors utiliser les mocks pour réaliser cette tâche.

Pour cela je crée aussi deux méthodes :

- **start** : qui est une fixture qui se charge de :
  - créer le mock pour l'interface IPokemonMetadata et y affecter un comportement lorsque la méthode getPokemonMetadata sera appelée
  - créer quelques instances de PokemonMetadata
- **testGetPokemonMetadata** : qui va utiliser les mocks créés dans la méthode start pour tester la méthode getPokemonMetadata

Pour gérer le cas de l'exception j'ai ajouter le comportement ci-dessous dans la méthode start :

```
pmdp = Mockito.mock(IPokemonMetadataProvider.class);
Mockito.doThrow(new PokedexException("Index Invalide"))
               .when(pmdp)
               .getPokemonMetadata(Mockito.intThat(i -> i < 0 || i > 150));
```

Ainsi dans la méthode testGetPokemonMetadata, il fallait bien vérifier que lorsque nous appelons la méthode avec un index invalide qu'une exception était belle et bien retournée.

```
assertThrows(PokedexException.class, ()->{
          pmdp.getPokemonMetadata(160);
      });
```

3.0.4 Le usecase IPokemonTrainerFactoryTest

Le usecase IPokemonTrainerFactoryTest permet de tester l'interface IPokemonTrainerFactory et cette interface ne contient qu'une seule méthode du coup on aura qu'un seul test unitaire pour cette classe.

Ci dessous la signature de la méthode que contient l'interface IPokemonTrainerFactory.

```
PokemonTrainer createTrainer(String name, Team team, IPokedexFactory pokedexFactory);
```

Cette méthode prend en paramètre les configurations d'un pokemonTrainer puis un IPokedexFactory, crée le pokemonTrainer puis le retourne

De même que précédemment vu que nous n'avons pas son implémentation, nous allons alors utiliser les mocks pour réaliser cette tâche.

Pour cela je crée aussi deux méthodes :

- **start** : qui est une fixture qui se charge de :
  - créer un mock de IPokemonTrainerFactory et y affecter des comportements
  - créer un mock de IPokedexFactory
  - créer un mock de IPokedex
  - créer une instance de PokemonTrainer
- **testCreateTrainer** : qui utilise les mocks et instance précédemment créé pour effectuer le test

3.0.5 Le usecase IPokedexTest

Le usecase IPokedexTest permet de tester l'interface IPokedex et cette interface contient 5 méthodes du coup on aura autant de usecase que de méthodes.

Ci-dessous les signatures des méthodes contenues dans l'interface IPokedexTest.

```
int size();
int addPokemon(Pokemon pokemon);
Pokemon getPokemon(int id) throws PokedexException;
List<Pokemon> getPokemons();
List<Pokemon> getPokemons(Comparator<Pokemon> order);
```

De même que précédemment vu que nous n'avons pas son implémentation, nous allons alors utiliser les mocks pour réaliser cette tâche.

Pour cela je crée aussi 6 méthodes :

- **start** : qui est une fixture qui se charge de :
  - créer un mock pour IPokedex et y affecter des comportements
  - créer des instances de Pokemon
  - créer une instance de Comparator<Pokemon>
- **TestSize** : qui va tester la méthode size
- **TestGetPokemon** : qui va tester la méthode getPokemon
- **TestGetPokemons** : qui va tester la méthode getPokemon
- **TestGetPokemonsWithComparator** : qui va tester la méthode getPokemons(Comparator<Pokemon> order)

4 Conclusion

Dans cet tp, il était question de tester les differentes classes et interfaces de notre projet à travers les tests unitaires et les doublures.

Ce travail a été réalisé en utilisant les frameworks **JUnit** et **Mockito**.

Références

[1] Daniel SALAS. *TP3 : Pokéunit*. Université Avignon. 2023. url : https://github.com/Youkoulanda/ceri-m1-techniques-de-test/blob/master/TPs/TP3.md.

[2] LAMAH Michel Marie. *TP3 : Pokéunit [master]*. Université Avignon. 2023. url : https://github.com/MichelMarieLamah13/ceri-m1-techniques-de-test/tree/master.

[3] LAMAH Michel Marie. *TP3 : Pokéunit [TP03]*. Université Avignon. 2023. url : https://github.com/MichelMarieLamah13/ceri-m1-techniques-de-test/tree/TP03.