Besson Fanny

Les principes SOLID
PokédexProject

1.Single Purpose Responsability

Afin de s'assurer que chaque classe n'a qu'une seule responsabilité, on utilise le modèle Model View Controller.

La classe *Pokemon* est le modèle, elle vous nous servir à définir le type de données et les données qui définissent un pokémon.
L'interface *PokemonController* et les classes *PokemonControllerHTTP* et *PokemonControllerSQLLite* qui l'implémentent font parti du Controller. Elles permettent de mettre en lien le modèle et son visionnage. Ici, elles ont pour but de récupérer les informations d'un pokémon selon l'identifiant fourni, et de le stocker dans un objet Pokémon.
La classe *PokemonView* est le View. Elle ne s'occupe que d'afficher les données d'un Pokémon, comme si on avait affaire à un pokédex.

Pokédex est notre classe exécutable.

A chaque classe est donc associée une tâche précise, et une seule.

2. Open-Closed

En rajoutant l'accès à une base de données locale et l'information de *description*, on va devoir modifier certaines parties du code.

On doit tout d'abord ajouter la variable *description* à la classe *Pokémon* et créer le constructeur associé mais on ne modifie pas le reste de la classe, on respecte donc le principe de Open-Closed.

Dans *PokemonView*, on ne faisait pas directement appel à la classe *PokemonControllerHTTP* pour recueillir les informations du pokémon mais on faisait appel à l'interface *PokemonController*, qui ne se préoccupe pas de la méthode utilisé pour récupérer les informations.

```java
public class Pokemon {
    //This class is to model the data of a Pokemon
    //Model in the MVC model

    private String id;
    private String name;
    private Long weight;
    private Long height;
    private String description;

    //constructor
    public Pokemon(String id, String name, Long weight, Long height) {
        this.id = id;
        this.name = name;
        this.weight = weight;
        this.height = height;
    }

    public Pokemon(String id, String name, Long weight, Long height, String description) {
        this.id = id;
        this.name = name;
        this.weight = weight;
        this.height = height;
        this.description = description;
    }
```

On n'a donc pas besoin de modifier *PokemonControllerHTTP*, on créée seulement une nouvelle classe *PokemonControllerSQLLite* qui implémente *PokemonController* et possède l'url de la base de données *url* comme variable et qui ré-écrit la méthode *getPokémon* pour qu'on aille récupérer les informations depuis la base de données locales. De nouveau, on a respecté le principe Open-Closed.

```
2
3   import java.sql.*;
4
5   public class PokemonControllerSQLLite implements PokemonController {
6
7       private String url; //
8
9       //Constructor
10      public PokemonControllerSQLLite(String url) {
11          this.url = url;
12      }
13
14      @Override
15      //getPokemon take a Pokemon pokemon and the id it has in the database stored in url,
16      //and return the pokemon after having update its variables (name, height, weight, description)
17      //to match those of the database
18      public Pokemon getPokemon(String id, Pokemon pokemon) {
19          Connection conn = null;
20          try {
21
22              // create a connection to the database
23              conn = DriverManager.getConnection(url);
24              System.out.println("Connection to SQLite has been established.");
25
26              //get the informations once the connection is established, with a query
```

Toutefois, dans la méthode *generatedView* de la classe *PokemonView*, on a dû rajouter quelques lignes (l.30-32) pour prendre en compte le fait qu'il puisse y avoir -ou non- un champ de *description* pour le pokémon. Toutefois, le rajout de cette ligne n'impacte pas les autres classes. On a respecté le principe Open-Closed puisque nos rajouts, ne viennent pas supprimer ce qui avait été fait.

```
18  public void generateView(String id){
19
20      //Get the information of the pokemon with the id id from the database
21      setPokemon(pokemonController.getPokemon(id,this.pokemon));
22
23      //Show the informations
24      System.out.println("============================");
25      System.out.println("Pokemon #"+id);
26      System.out.println("Nom : " + pokemon.getName());
27      System.out.println("Taille : " + pokemon.getHeight());
28      System.out.println("Poids : " + pokemon.getWeight());
29      //Add, the description variable of a Pokemon can be empty so we need to make sure it is not before
30      if (!pokemon.getDescription().isBlank()){
31          System.out.println("Description : " + pokemon.getDescription());
32      }
33      System.out.println("============================");
34  }
```

3. Liskov Substitution

Dans ce projet, je n'ai pas utiliser de relations d'héritages, le principe de Liskov Substitution ne peut donc pas être appliqué.

J'ai en effet préférer utiliser une interface *PokemonController*, plutôt que de faire en sorte que *PokemonControllerSQLLite* hérite de *PokemonControllerHTTP*.

Dans le cas où *PokemonControllerSQLLite* aurait hérité de *PokemonControllerHTTP*, on aurait bien eu une réécriture de la méthode *getPokemon(String id, Pokemon pokemon)* mais l'ajout de la variable *url*, étant nécessaire à cette méthode (puisque on a besoin de savoir où est stocké le fichier de la base de données), n'aurait pas permit d'utiliser une instance de *PokemonControllerSQLLite* comme une instance de *PokemonControllerHTTP* et on aurait par conséquent pas respecter le principe de Liskov Substitution. Ou alors, on aurait du rajouter une variable *url* également dans *PokemonControllerHTPP* mais elle n'aurait pas été utilisé et cela aurait modifié cette classe après son écriture, et je souhaitais ne pas la modifier pour rester proche du principe Open-Closed.
Pour ces raisons, j'ai préféré utiliser une interface, plutôt qu'une relation d'héritage.

```java
public interface PokemonController {
    //Controller in the MVC model

    //getPokemon take a Pokemon pokemon and the id it has in the database,
    //and return the pokemon after having update its variables
    //to match those of the database
    public Pokemon getPokemon(String id, Pokemon pokemon);

}
```

```java
public class PokemonControllerHTTP implements PokemonController {

```

```java
public class PokemonControllerSQLLite implements PokemonController {
```

4. Interface Segregation

Ici, on a utilisé une seule interface *PokemonController* qui utilise une méthode *getPokemon* pour récupérer l'entièreté des informations du pokémon. Le principe Interface Segregation n'a donc pas été mis en œuvre.

Toutefois, on aurait pu imaginer ne vouloir afficher que le nom du Pokémon, ou seulement le nom et son poids. Dans ce cas, il aurait fallu faire une interface différente pour récupérer chaque information séparément :
-une interface *PokemonControllerName* qui a pour méthode *getPokemonName(String id, Pokemon pokemon)* et ne récupère que le nom du Pokémon
- une interface *PokemonControllerWeight* qui a pour méthode *getPokemonWeight(String id, Pokemon pokemon)* et ne récupère que le poids du Pokémon
- une interface *PokemonControllerDescription* qui a pour méthode *getPokemonDescription(String id, Pokemon pokemon)* et ne récupère que la description du Pokémon
-…

Dans ce cas, la classe *PokemonControllerHTTP* aurait implémenté les interfaces *PokemonControllerName, PokemonControllerWeight* et *PokemonControllerHeight*. La classe *PokemonControllerSQLLite* aurait implémenté les mêmes interfaces et *PokemonControllerDescription*.

5. Dependency Inversion Principle

```
2
3   public class PokemonView {
4       //View in the MVC model
5       //equivalent to a Pokemon view
6
7
8       private PokemonController pokemonController;
9       private Pokemon pokemon;
10
11      //Constructor
12      public PokemonView( PokemonController pokemonController) {
13          this.pokemonController=pokemonController;
14          this.pokemon= new Pokemon( id: "", name: "", weight: null, height: null, description: "");
15      }
16
17      //generateView take the id of the pokemon we want to show, and show its information as if it was on a pokedex.
18      public void generateView(String id){
19
20          //Get the information of the pokemon with the id id from the database
21          setPokemon(pokemonController.getPokemon(id,this.pokemon));
22
23          //Show the informations
24          System.out.println("============================");
25          System.out.println("Pokemon #"+id);
26          System.out.println("Nom : " + pokemon.getName());
27          System.out.println("Taille : " + pokemon.getHeight());
28          System.out.println("Poids : " + pokemon.getWeight());
29          //Add, the description variable of a Pokemon can be empty so we need to make sure it is not before showing it
30          if (!pokemon.getDescription().isBlank()){
31              System.out.println("Description : " + pokemon.getDescription());
32          }
33          System.out.println("============================");
34      }
```

La classe *PokemonView* fait appel à une instance de l'interface (abstraite) *PokemonController* (l.8) pour exécuter la requête et récupérer les informations à afficher du Pokémon. Cette classe ne se préoccupe pas de la façon dont est exécutée la requête (si c'est une requête SQL ou HTTP), et on peut lui fournir n'importe quelle instance de plus bas niveau (*PokemonControllerHTTP* ou *PokemonControllerSQLLite*).

Cela se voit notamment au niveau des tests unitaires effectuer sur cette classe.

```java
import org.junit.Test;
import org.assertj.core.api.Assertions;

public class PokemonViewTest {

    @Test
    //Test with the HTTPrequest
    public void generateViewHTTP() {
        //create the appropriate PokemonView instance
        PokemonControllerHTTP pokemonController = new PokemonControllerHTTP();
        PokemonView pokemonView = new PokemonView(pokemonController);
        pokemonView.generateView( id: "1");

        //Tests to be sure that the info we get are the one we wanted
        Assertions.assertThat(pokemonView.getPokemon().getName()).isEqualTo("bulbasaur");
        Assertions.assertThat(pokemonView.getPokemon().getWeight()).isEqualTo(69L);
        Assertions.assertThat(pokemonView.getPokemon().getHeight()).isEqualTo(7L);
        Assertions.assertThat(pokemonView.getPokemon().getDescription()).isEqualTo("");

    }
```

```java
    @Test
    //Test with the SQL query
    public void generateViewSQLLite() {
        //Create the appropriate PokemonView instance
        PokemonControllerSQLLite pokemonController = new PokemonControllerSQLLite( url: "jdbc:sqlite:/tmp/pokemondatabase.sqlite
        PokemonView pokemonView = new PokemonView(pokemonController);
        pokemonView.generateView( id: "1");

        //Tests to be sure the info we get are the one we wanted
        Assertions.assertThat(pokemonView.getPokemon().getName()).isEqualTo("Bulbizarre");
        Assertions.assertThat(pokemonView.getPokemon().getWeight()).isEqualTo(69L);
        Assertions.assertThat(pokemonView.getPokemon().getHeight()).isEqualTo(7L);

        //The test on the description return Failed as there is a problem with the accent and the encoding of the text, howev
        Assertions.assertThat(pokemonView.getPokemon().getDescription()).isEqualTo("Il a une étrange graine plantée sur son d

    }
```

On peut écrire un test avec une instance de la classe *PokemonControllerHTTP* dans le constructeur de la classe *PokemonView* et tester si les informations récupérées (qui seront celles afficher par la méthode *generateView*) correspondent à celles voulues.

On peut de même écrire un autre test avec une instance de la classe *PokemonControllerSQLLite* dans le constructeur de la classe *PokemonView*. Le reste du test étant identique à l'exception de cette ligne (et du test sur la variable *description* qui aura ici une information de part la nature de la base de donnée qu'on utilise), on a respecté le princie de Dependency Inversion.