Ecole des Mines de Saint Etienne

Toolbox 3 : Ingénierie et interopérabilité des systèmes informatiques

Rapport TP Mise en pratique des principes SOLID

Présenté par : Joe AL HASROUNI

L'application doit être fonctionnelle, et pouvoir être lancée facilement en ligne de commande (des instructions claires sont dans un README file). L'application à été développer sur Windows, l'établissement d'une connexion avec une base de données ne marchait pas. Si c'est encore le cas quand vous essayer d'exécuter le code vous pouvez essayer le terminal déjà présent dans l'IDE IntelliJ.

Explication des principes SOLID mis en œuvre dans l'application.

S: Single-purpose Responsibility

Le principe Single-prupose Responsibility est bien implémenté dans notre application Pokedex. En effet en jetant un coup d'œil sur les classes présents et leurs nominations, on constate rapidement que les taches sont bien divisées entre les classes. Le modèle MVC est implémenté.

Chaque classe ici possède une seule fonctionnalité :

- IPokemonContoller : Interface pour appeler les classes nécessaires pour construire l'objet PokemonCharacter
  - IPokemonControllerAPI : Appel les classes pour former l'objet Pokémon de l'API.
    - HTTPRequest : Former l'URL correspondant pour récupérer les données désirées.
    - HTTPResponse : Recevoir la réponse JSON et la mettre dans un string.
    - ParseJson : Examiner le string Json et former l'objet Pokémon.
  - IPokemonControllerSQL : Appel les classes pour former l'objet Pokémon des données SQL.
    - SQLiteConnection : Etablir une connexion SQL avec la base de donnees desirée.
    - SQLGetData : Récupérer les données nécessaires et former l'objet Pokémon.
- IPokemonViewer : Interface pour afficher les objets Pokémon.
  - PokemonViewerAPI : Afficher l'ID, le nom, la taille et le poids du Pokémon.
  - PokemonViewerSQL : Afficher l'ID, le nom, la taille, le poids et la description du Pokémon.
- PokemonCharacter : former le modèle du personnage Pokémon.
  - PokemonCharacterSQL : Hérite de la classe PokemonCharacter et ajoute le membre description.

O: Open-Closed

En premier temps, l'application consistait juste de récupérer les données de l'API. Le code était ouvert pour ajouter des nouvelles fonctionnalités. Pour cela il est très simple maintenant d'ajouter l'option de traiter les données d'une base de données locale.

En effet la structure du code permet cela car juste en implémentant l'interface IPokemonController par une autre classe spécifier pour contrôler le partie SQL et même chose pour l'interface IPokemonViewer.

Et encore la classe PokemonCharacterSQL hérite de la classe PokemonCharacter.

```
public class PokemonControllerSQL implements IPokemonController
public class PokemonViewerSQL implements IPokemonViewer
public class PokemonCharacterSQL extends PokemonCharacter
```

L: Liskov Substitution

L'ajout de classes héritées ne devrait pas entraver le fonctionnement d'un système déjà existant. Un bon exemple du respect du principe de substitution de Liskov dans notre code est le suivant :

```
public class ParseJson {

   //Setting the Json string
   private String jsonResponse;
   public void setJsonResponse(String jsonResponse) {
      this.jsonResponse = jsonResponse;
   }

   //Instance of the class PokemonCharacter
   private PokemonCharacter myCharacter;
   public PokemonCharacter getMyCharacter() {
      return myCharacter;
   }

   //Constructor
   ParseJson(String jsonResponse)
   {
      myCharacter = new PokemonCharacterSQL();
      setJsonResponse(jsonResponse);
   }
...
```

Dans le bloque du code ci-dessus de la classe ParseJson, on remarque que cette classe possède un membre objet privé de type PokemonCharacter. Dans le constructeur de cette classe, cette instance est instanciée de la classe PokemonCharacterSQL qui est une sous-classe de la classe base PokemonCharacter. Et c'est possible puisque la sous-classe n'a pas des conditions plus fortes que celles de sa superclasse.

D: Dependancy Inversion

Les classes PokemonViewerAPI et PokemonViewerSQL font un bon exemple pour le principe Depandancy Inversion. En effet le principe Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.

C'est vrai ici que c'est deux classes affiche les informations récupérer d'une API ou d'une base de données locale. Mais c'est deux classes peuvent marcher sans problème sans la source de données puisque les méthodes définis dans ces classes dépendent seulement de l'objet fournit. Ce principe est vérifié par l'application d'un test unitaire sur une de ces deux classes, soit PokemonViewerSQL. Dans ce test, on donne à la classe un objet instancié par défaut de la classe PokemonCharacterSQL et le test réussi.

```
public class PokemonViewerAPI implements IPokemonViewer {
    public static void printPokemon(PokemonCharacter myCharacter)
    {
. . .
```

Test Unitaire:

```
public class PokemonViewerSQLTest extends TestCase {
    public void testPrintPokemon() {
        PokemonCharacterSQL myCharacterTest = new PokemonCharacterSQL();
        Assert.assertEquals("1 bulbasaur 7
69",PokemonViewerSQL.printPokemon(myCharacterTest));
    }
}
```

I : Interface Segregate

Avec le sujet actuel, il est probable que le principe Interface Segregation n'a pas été mis en œuvre.
Imaginons un scenario le cas ou un utilisateur veut afficher les caractéristiques du Pokémon dans un fichier HTML comme une fonctionnalité additionnelle sur l'application.
On peut toujours modifier l'interface IPokemonViewer en ajoutant une méthode pour générer le fichier HTML. Mais comme ça on ne respecte pas le principe Interface Segregation car il est maintenant obligatoire d'implémenter cette méthode dans toutes les classes venant de l'interface.
Pour cela le principe Interface Segregate nous dit que pour ajouter une fonctionnalité qui n'est pas utilisé par tous les utilisateurs, il est mieux d'ajouter une nouvelle interface spécifiée pour cette tâche précise. Par exemple IPokemonViewerHTML qui sera ultérieurement implémentée par PolemonViewerHTML_API et PokemonViewerHTML_SQL.