Rapport du projet Pokedex

Bonnes pratiques de développement logiciel

Ghizlane ZEHNINE
ghizlane.zehnine@etu.emse.fr

Ingénierie et Interopérabilité
des Systèmes Informatiques

November 28, 2020

Contents

1 Introduction

Podédex est une application qui affiche les données (nom, taille, poids et description) d'un pokémon demandé en utilisant soit l'api Pokeapi ou en accédant à la base de donnée locale sqLite.
C'est un travail pratique des connaissances apprises au cours "Bonnes pratiques de développement logiciel" et des principes SOLID dont nous allons expliquer la mise en oeuvre dans notre application.

2 Les principes SOLID

En programmation orientée objet, SOLID est un acronyme qui regroupe cinq principes de conception destinés à produire des architectures logicielles plus compréhensibles, flexibles et maintenables.

2.1 Responsabilité unique (*Single responsibility principle*)

2.1.1 Définition

Une classe, une fonction ou une méthode doit avoir une et une seule responsabilité.

2.1.2 Mise en oeuvre

Avant on avez une seule classe SQLLiteExample qui s'occupe de la connexion avec la BD et des requêtes SQL ainsi que de l'affichage des résultat.
Maintenant, en utilisant le principe *Single responsibility* on a créer pour chaque objectif une classe:

- Pour établir la connexion avec la base de données locale sqlite, on a créer une classe DBconnection qui n'a pour objectif que d'établir cette connexion.

```java
package com.example.pokedex;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBconnection implements DBconnectionInterface {
    private String url;

    public DBconnection(String url) {
        this.url = "jdbc:sqlite:"+url;
    }
    public Connection establish() {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(this.url);
            System.out.println("Connection to SQLite has been established.");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return  conn;
    }
    public String getUrl() { return url; }
    public void setUrl(String url) { this.url = url; }
}
```

Figure 1: Classe DBconnection

- Pour gérer les requêtes SQL et les résultats des requêtes on a créer une classe PokemonDAO.

```java
package com.example.pokedex;

import java.nio.charset.StandardCharsets;
import java.sql.*;

public class PokemonDAO {
    PokemonDetailed pokemon;
    public PokemonDetailed get(int id, DBconnectionInterface db){
        PreparedStatement stmt  = null;
        try {
            stmt = db.establish().prepareStatement( sql: "SELECT name, description , " +
                    "height , weight FROM pokemons WHERE id = ?");
            stmt.setInt( parameterIndex: 1, id);
            ResultSet rs = stmt.executeQuery();
            rs.next();
            String descriptionWithEncodingProblem = rs.getString( columnLabel: "description");
            String description = new String(descriptionWithEncodingProblem.getBytes(StandardCharsets.UTF_8));
            pokemon = new PokemonDetailed(rs.getString( columnLabel: "name"),
                    id, rs.getString( columnLabel: "weight"), rs.getString( columnLabel: "height"),description );
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return pokemon;
    }
}
```

Figure 2: Classe PokemonDAO

- Pour gérer le pokémon comme objet, on a créé une classe Pokemon et une classe PokemonDetailed (qu'on va voir dans le prochain principe).

- Ensuite pour pouvoir controler ces classes citées précédemment, on a la classe SQLLite qui fait appel aux méthodes des autres classes pour récupérer les données du Pokemon à partir de la BD.

```java
package com.example.pokedex;

public class SQLLite {
    public static PokemonDetailed run(int id, String url){

        DBconnection dbConnection = new DBconnection(url);
        dbConnection.establish();

        PokemonDAO pokemonDAO = new PokemonDAO();
        PokemonDetailed pokemon;
        pokemon = pokemonDAO.get(id , dbConnection);

        return pokemon;

    }
}
```

Figure 3: Classe SQLLite

- Puis pour l'affichage des données à partir de l'API, on a une classe HTTPRequest qui s'occupe de l'envoie de requête HTTP et la réception des réponses sous format json.

```java
package com.example.pokedex;
import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import java.io.IOException;

public class HTTPRequest {
    private CloseableHttpClient httpClient;
    private HttpGet request;

    public HTTPRequest() { this.httpClient = HttpClientBuilder.create().build(); }
    public String get(String uri){
        String jsonResponse = "";
        HttpGet request = new HttpGet(uri);
        request.addHeader( name: "content-type",  value: "application/json");
        HttpResponse result = null;
        try {
            result = httpClient.execute(request);
            jsonResponse = EntityUtils.toString(result.getEntity(),  defaultCharset: "UTF-8");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return jsonResponse;
    }
}
```

Figure 4: Classe HTTPRequest

- Après la réception des données sous format json on les convertit en objet Pokemon à l'aide de la classe PokemonFromJSON.

```java
import org.json.simple.JSONObject;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class PokemonFromJSON {
    Pokemon pokemon;
    public Pokemon parse(String json , int id){
        JSONParser parser = new JSONParser();
        Object resultObject = null;
        try {
            resultObject = parser.parse(json);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        if (resultObject instanceof JSONObject) {
            JSONObject obj =(JSONObject)resultObject;
            pokemon = new Pokemon(obj.get("name").toString() , id ,
                    obj.get("weight").toString() , obj.get("height").toString());
        } else {
            System.err.println("Error, we expected a JSON Object from the API");
        }
        return pokemon;
    }
}
```

Figure 5: Classe PokemonFromJSON

- Finalement, on a la classe HTTPRequestController qui fait appel aux autres classes pour récupérer les données du Pokemon à partir de l'API et de les stocker dans un objet Pokemon.

```java
package com.example.pokedex;

public class HTTPRequestController {
    public static Pokemon run(int id){
        HTTPRequest httpRequest = new HTTPRequest();
        String response = "";
        response = httpRequest.get("https://pokeapi.co/api/v2/pokemon/"+id);

        PokemonFromJSON pokemonFromJSON = new PokemonFromJSON();
        Pokemon pokemon;
        pokemon = pokemonFromJSON.parse(response , id);

        return pokemon;
    }
}
```

Figure 6: Classe HTTPRequestController

2.2 Ouvert/fermé (*Open/closed principle*)

2.2.1 Définition

Une entité applicative (class, fonction, module ...) doit être ouverte à l'extension, mais fermée à la modification.

2.2.2 Mise en oeuvre

Au début, notre application affichait les données à partir de l'api seulement. Donc on a créer une classe Pokemon qui a comme attributs : nom, taille et poids. Ensuite on a ajouté une nouvelle fonctionnalité de pouvoir obtenir les données à partir d'une base de données locale si l'utilisateur le souhaite. Cette base de données possède une information supplémentaire dans son jeu de donnée, la description du pokémon. Donc pour respecter le principe *Open/Closed* on a créer une nouvelle classe PokemonDetailed qui hérite de la classe Pokemon et qui possède l'attribut description de plus.

```java
package com.example.pokedex;

public class Pokemon {
    private String name;
    private int id;
    private String weight;
    private String height;

    public Pokemon(String name, int id, String weight, String height) {
        this.name = name;
        this.id = id;
        this.weight = weight;
        this.height = height;
    }
    public String toString(){
        return "Pokémon #"+this.getId()+"\nNom : "+this.getName()+"\nTaille : " +
                this.getHeight()+"\nPoids : "+this.getWeight();
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getWeight() { return weight; }
    public void setWeight(String weight) { this.weight = weight; }
    public String getHeight() { return height; }
    public void setHeight(String height) { this.height = height; }
}
```

Figure 7: Classe Pokemon

```java
package com.example.pokedex;

public class PokemonDetailed extends Pokemon{
    private String description;
    public PokemonDetailed(String name, int id, String weight, String height,
                           String description) {
        super(name, id, weight, height);
        this.description = description;
    }

    public String toString(){
        return "Pokémon #"+this.getId()+"\nNom : "+this.getName()+
                "\nTaille : "+ this.getHeight()+"\nPoids : "+this.getWeight()+
                "\nDescription :"+this.getDescription();
    }
    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }
}
```

Figure 8: Classe PokemonDetailed

2.3 Substitution de Liskov (*Liskov substitution principle*)

2.3.1 Définition

Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme.

2.3.2 Mise en oeuvre

Dans notre méthode main de la classe Pokedex, on déclare un objet Pokemon et selon le choix de l'utilisateur (API ou Base de donnée) on lui affecte le résultat de nos contrôleur: qui est de type PokemonDetailed dans le cas de SQLLite et de type Pokemon dans le cas de HTTPRequestController.

```java
// SQLLite.run() retourne PokemonDetailed
pokemon = SQLLite.run(Integer.parseInt(args[0]) , args[1]);
//HttpRequestController.run() retourne Pokemon
pokemon = HTTPRequestController.run(Integer.parseInt(args[0]));
```

Figure 9: Exemple du principe substitution de Liskov

2.4 Inversion des dépendances (*Dependency inversion principle*)

2.4.1 Définition

Il faut dépendre des abstractions, pas des implémentations.

2.4.2 Mise en oeuvre

Dans la classe PokemonDAO, on a besoin de récupérer la connection établie par la classe DBconnection. Ainsi notre classe PokemonDAO dépend de DBconnection. Donc pour respecter le principe *Dependency inversion*, on a créer une intérface DBconnectionInterface implémentée par DBconnection, ensuite on modifié la classe PokemonDAO afin qu'elle dépende de l'interface et non pas de son implémentation.

```java
package com.example.pokedex;

import java.sql.Connection;

public interface DBconnectionInterface {
    public Connection establish();
}
```

```java
package com.example.pokedex;

import java.nio.charset.StandardCharsets;
import java.sql.*;

public class PokemonDAO {
    PokemonDetailed pokemon;
    public PokemonDetailed get(int id, DBconnectionInterface db){
        PreparedStatement stmt  = null;
        try {
            stmt = db.establish().prepareStatement( sql: "SELECT name, description , " +
                    "height , weight FROM pokemons WHERE id = ?");
            stmt.setInt( parameterIndex: 1, id);
            ResultSet rs = stmt.executeQuery();
            rs.next();
            String descriptionWithEncodingProblem = rs.getString( columnLabel: "description");
            String description = new String(descriptionWithEncodingProblem.getBytes(StandardCharsets.UTF_8));
            pokemon = new PokemonDetailed(rs.getString( columnLabel: "name"),
                    id, rs.getString( columnLabel: "weight"), rs.getString( columnLabel: "height"),description );
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return pokemon;
    }
}
```

Figure 10: Classe PokemonDAO qui dépend de l'interface DBconnectionInterface