SOEN 6441 – Section W

Advanced Programming Practices

Instructor: Dr. C. Constantinides, P.Eng.

Project Report

Submitted By:

Student Name: Sonam Sonam

Student Id: 40205470

Partner Student name: Vikram Singh Brahm

Student Id: 40241024

Contents

1. Introduction

This is an **Advanced Programming Practices (SOEN 6441)** project which has below requirements:

- Use any language (or combination of languages) to develop a system that can read API /JSON data and produce results to some application.
- For the purpose of this exercise, you should aim at creating a small and manageable local database based on the downloaded data.
- You may not use any frameworks (other than, possibly, for the presentation of data).
- Your system should allow basic parameterized queries.

This project uses Java programming language to develop the system. We have selected an API which provides data in JSON format. We have parsed that data and created a local database which stores the data in relational model. An application is then developed which works on this data and presents it to the end user.

2. Technologies Used

2.1. API

- SWAPI, URL: https://swapi.dev/
- This API provides the data about Star Wars movies and characters in JSON format.
- There are details about People (Characters), movies, planets, species etc.
- For the sake of simplicity, we are not taking the entire data that this API provides. Rather, we take a subset of the data and using Java, we parse that data and create a local database in MySQL workbench.

2.2. Back-end

- MySQL workbench

2.3. Application Language

- Java

2.4. Front-end

- Java Server Pages & HTML
- For presenting the data, we are using Java Server Pages to show the basic queries being made on the database.

2.5. Testing Framework

- JUnit

3.API Details

Below is the data which we are capturing from SWAPI API and storing the same in relational model.

Characters/People

Provided by below URL

https://swapi.dev/api/people/1/

Characters in Star Wars movie which we are calling here as "People". We are collecting below data about People

People Id: Unique ID for every character

Name: Name of the character

Species: Species Id which the character belongs to

Height: Height of the character

Home World: Character's home world planet's Id

Birth Year: Character's Birth year

Gender: Gender details

Eye Color: Color of the character's eye

Hair Color: Color of character's hair

Skin Color: Color of character's skin

Movies

Provided by below URL

https://swapi.dev/api/films/2/

Movies table stores below data about each movie

Movie Id: Unique Id for every movie

Title: Movie's title

Producers: Producers of the movie

Director: Director of movie

Release date: Release date of the movie

Planets

Provided by below URL

https://swapi.dev/api/planets/1/

Planets table stores below data about each planet

Planet Id: Unique Id for every planet

Name: Name of the planet

Climate: Climate at the planet

Terrain: Terrain of planet

Population: Population of planet

Species

Provided by below URL

https://swapi.dev/api/species/1/

Species table stores below data about each species

Species Id: Unique Id for every species

Name: name of species

Classification: classification of the species

Language: language of the species

Designation: Designation of species

Home world: Home world of species

4. Patterns overview

4.1. Structural Patterns

- Simple mapping with Identity field
  - We have mapped each table to one class and added an ID field to uniquely identify a record or object
- Association Table mapping
  - We had many to many mapping between two entities where we implemented association table

4.2. Data Source Architectural Patterns

- Table Data Gateway
  - We are using TDGs to separate the application from "Access to resources".
  - All the database queries are written in TDG classes which interact with the database and return the corresponding objects to the application.
  - Application then passes this information to the presentation system.

4.3. Design Patterns

- Singleton
  - This project also includes a Logger class which logs the events to a log file. This makes debugging easy and helps in backtracking if there is any error or issue.
  - Logger class has been implemented using Singleton design pattern because at a time, we must have a single instance of logger which any class could use to log its messages.

5. Patterns in Detail

5.1. Structural Patterns

We have performed a simple mapping although there is no inheritance relationship, but simple mapping helps to keep data organized and manageable.

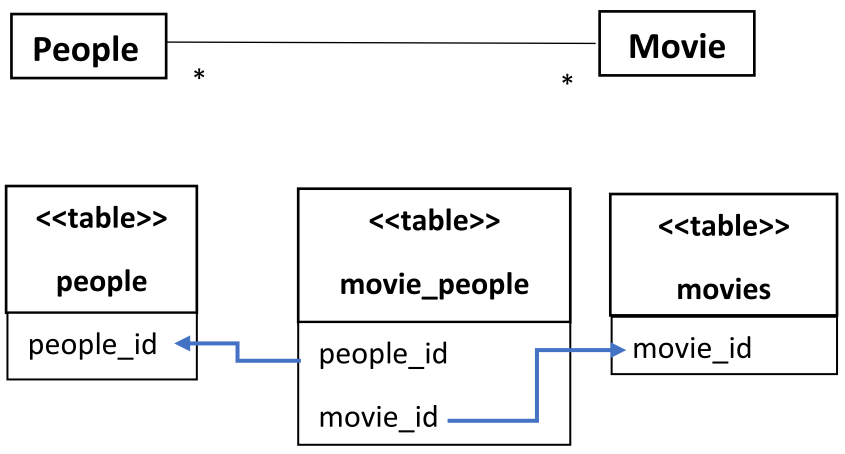
5.1.1. Simple Mapping with Identity field:

- Each Class has corresponding one table.
- People Class has a corresponding People table in relational model.
- Movie Class has a table named "movies" in relational model.
- Planet maps to "planets" table.
- Species class has a mapped table called "species".

Identity field is included in every table to uniquely identify all the records and to create a mapping between the tables using primary key foreign key relationship.

5.1.2. Association Table mapping:

Since there is a many to many relationships between People and Movies. So, People class can have multiple values for movies and similarly Movie class can have multiple values for People. So, we create an Association class called MoviePeopleAssociation and persist the movieId and peopleId in it. Therefore, there exists a corresponding table in relational structure named "movie_people" table.



5.2. Data Source Architectural Patterns

5.2.1. Table Data Gateway

We are employing a gateway which is used for encapsulating the access to resources. The gateway will act as an intermediary between our database and the application. The gateway we implement here is a Table Data Gateway. So, we have One TDG per class. Every class has its own TDG. Each class interacts with database through its corresponding TDG.

For People class, we have written a PeopleTableGateway class which encapsulates all the SQL queries posted on People table. For example, if we want to show details of a character with given identity, the method

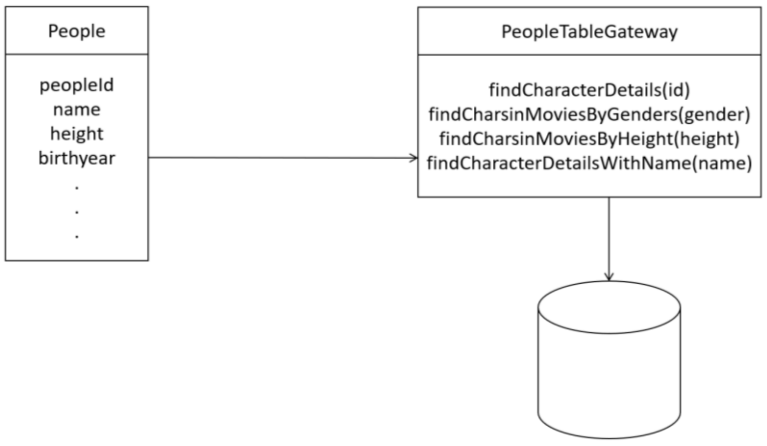
```
public People findCharacterDetails (int peopleId) {

……………….

}
```

provides the corresponding People object and we can access the instance variables using its getter methods. It encapsulates the query “select * from people where people_id = ?”.

Similarly, we have other methods which give the result set for other types of parameterized queries.

All the table data gateways extend from a base class called “TableDataGateway” which has username and password as its instance variables. There exists a static method connect(..) which creates a SQL connection object and returns it. This method is utilized by all the subclasses to connect to the database.

All TDGs are stateless, they just have a username and database to connect to the database, otherwise, they don’t maintain any state.


People
peopleId
name
height
birthyear
.
.
.
PeopleTableGateway
findCharacterDetails(id)
findCharsinMoviesByGenders(gender)
findCharsinMoviesByHeight(height)
findCharacterDetailsWithName(name)

5.3. Design Patterns

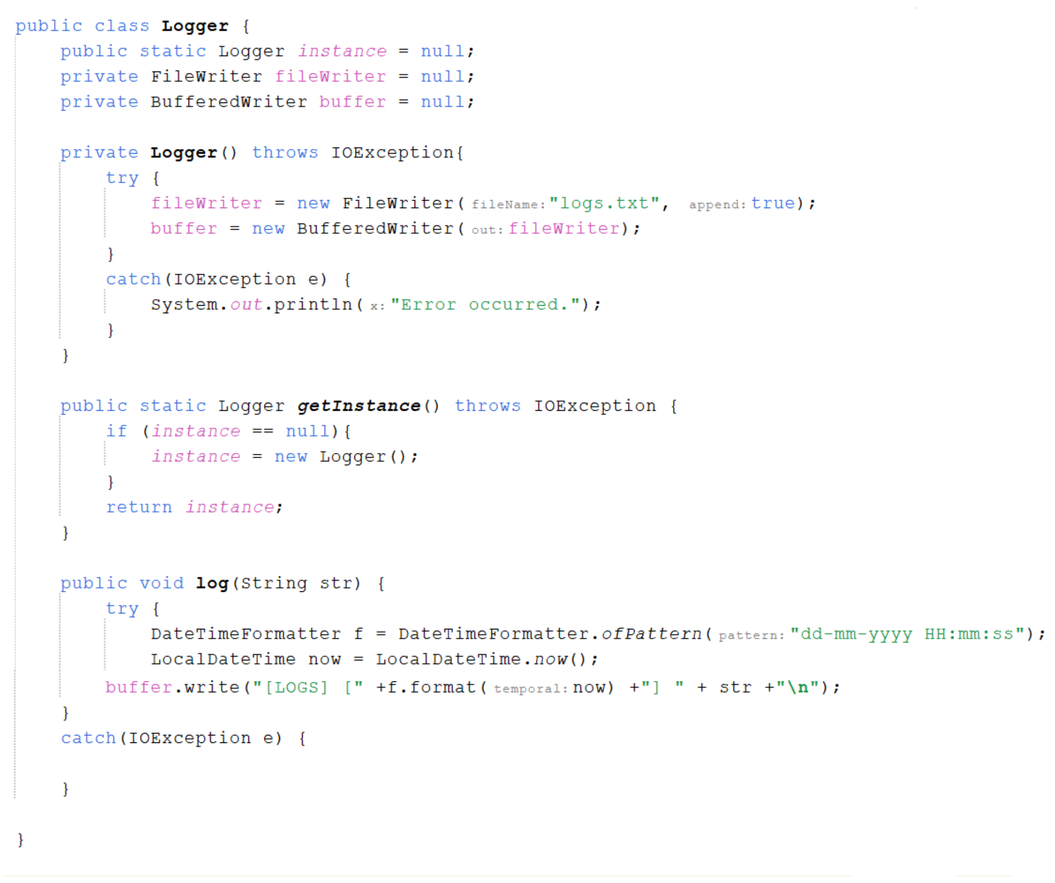
5.3.1. Singleton

We have added a Logger class which logs the events of Object creation, setting instance variables, getting data from database etc. All the events are captured in a text file. Logger class also logs the date and time of every event along with a message.

The Logger class has been implemented using Singleton pattern. The requirement here is to make sure we have a single instance of Logger class so that all the classes do not try to write together into the log file.

It has been implemented by having

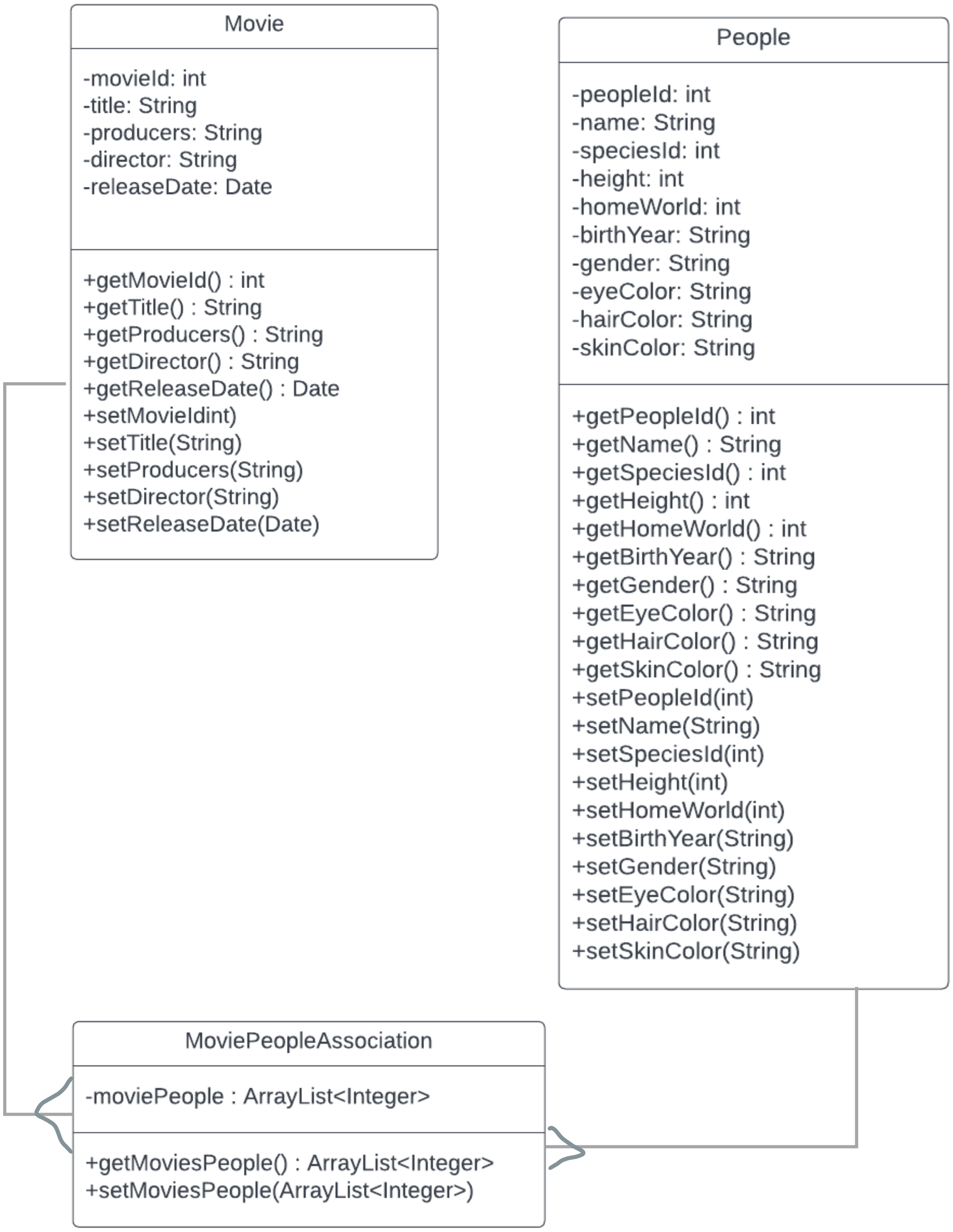
- a static instance of Logger class as instance variable
- making the default constructor private so as to hide the interface for creating an instance
- having a static method which calls the private constructor to construct an object and assign it to the static instance and return the static instance

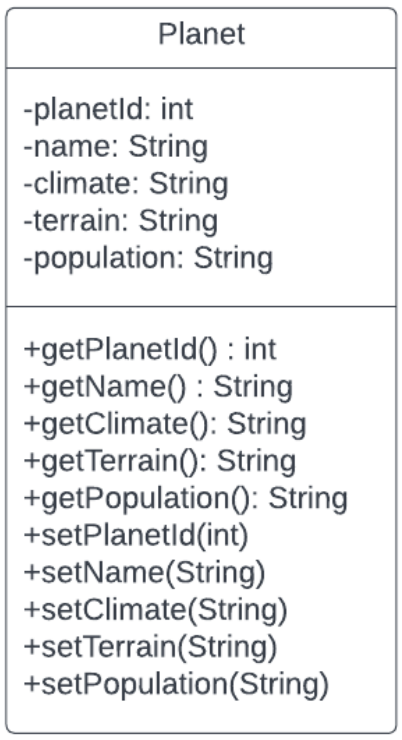


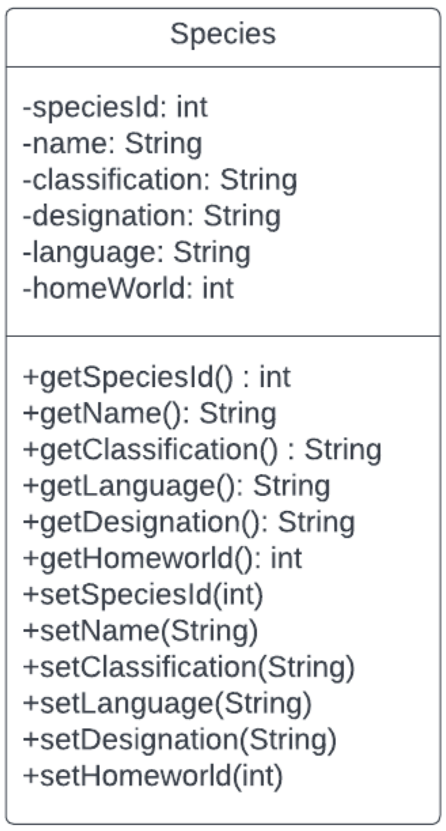
```java
public class Logger {
    public static Logger instance = null;
    private FileWriter fileWriter = null;
    private BufferedWriter buffer = null;

    private Logger() throws IOException{
        try {
            fileWriter = new FileWriter( fileName: "logs.txt",  append: true);
            buffer = new BufferedWriter( out: fileWriter);
        }
        catch(IOException e) {
            System.out.println( x: "Error occurred.");
        }
    }

    public static Logger getInstance() throws IOException {
        if (instance == null){
            instance = new Logger();
        }
        return instance;
    }

    public void log(String str) {
        try {
            DateTimeFormatter f = DateTimeFormatter.ofPattern( pattern: "dd-mm-yyyy HH:mm:ss");
            LocalDateTime now = LocalDateTime.now();
        buffer.write("[LOGS] [" +f.format( temporal: now) +"] " + str +"\n");
    }
    catch(IOException e) {

    }

}
```

6. Object Model

Application classes:




Planet
-planetId: int
-name: String
-climate: String
-terrain: String
-population: String
+getPlanetId() : int
+getName() : String
+getClimate(): String
+getTerrain(): String
+getPopulation(): String
+setPlanetId(int)
+setName(String)
+setClimate(String)
+setTerrain(String)
+setPopulation(String)


Species
-speciesId: int
-name: String
-classification: String
-designation: String
-language: String
-homeWorld: int
+getSpeciesId() : int
+getName(): String
+getClassification() : String
+getLanguage(): String
+getDesignation(): String
+getHomeworld(): int
+setSpeciesId(int)
+setName(String)
+setClassification(String)
+setLanguage(String)
+setDesignation(String)
+setHomeworld(int)

Logger:

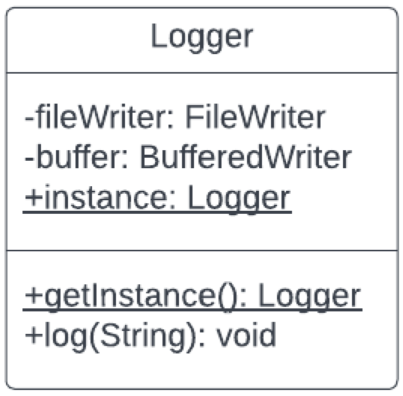
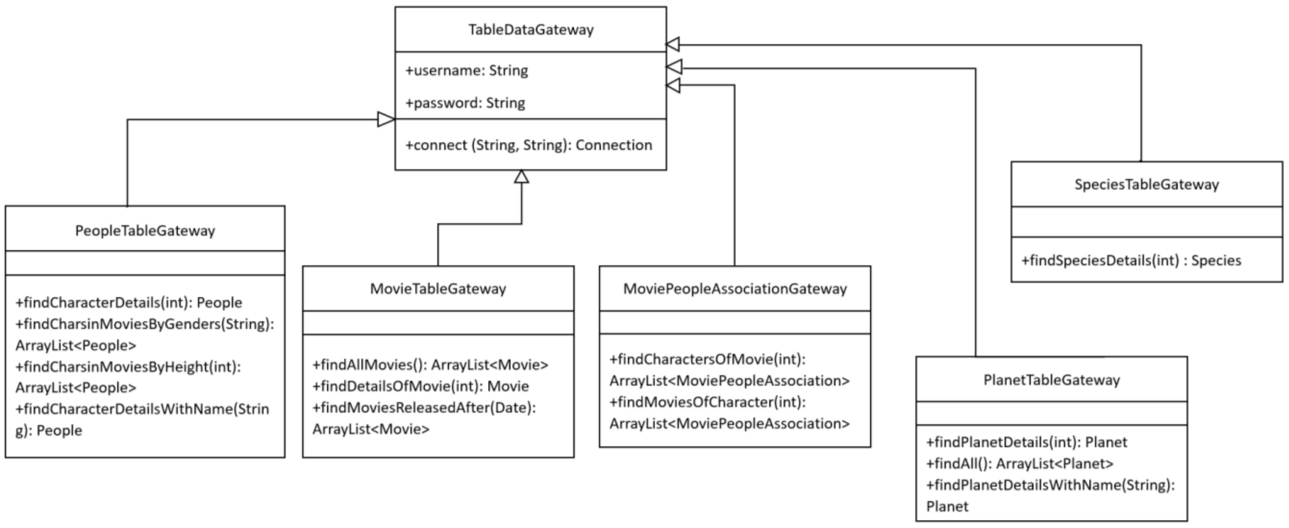

Logger
-fileWriter: FileWriter
-buffer: BufferedWriter
+instance: Logger
+getInstance(): Logger
+log(String): void

Table Data Gateways:


TableDataGateway
+username: String
+password: String
+connect (String, String): Connection
PeopleTableGateway
+findCharacterDetails(int): People
+findCharsinMoviesByGenders(String): ArrayList<People>
+findCharsinMoviesByHeight(int): ArrayList<People>
+findCharacterDetailsWithName(String): People
MovieTableGateway
+findAllMovies(): ArrayList<Movie>
+findDetailsOfMovie(int): Movie
+findMoviesReleasedAfter(Date): ArrayList<Movie>
MoviePeopleAssociationGateway
+findCharactersOfMovie(int): ArrayList<MoviePeopleAssociation>
+findMoviesOfCharacter(int): ArrayList<MoviePeopleAssociation>
PlanetTableGateway
+findPlanetDetails(int): Planet
+findAll(): ArrayList<Planet>
+findPlanetDetailsWithName(String): Planet
SpeciesTableGateway
+findSpeciesDetails(int) : Species

7. Relational Model

People Table

people_id	name	species	height	home_world	birth_year	gender	eye_color	hair_color	skin_color

Movies Table

movie_id	title	producers	director	release_date

Movie_People Table

While retrieving people details, we also get the list of movies in which each character has appeared. Same way, while extracting movie details, we get a list of characters which have appeared in a movie. So, we have a many to many relationships between People and Movie. As mentioned in Chapter 2, we have an association class between both.

So, our movie_people table looks as below

movie_id	People_id

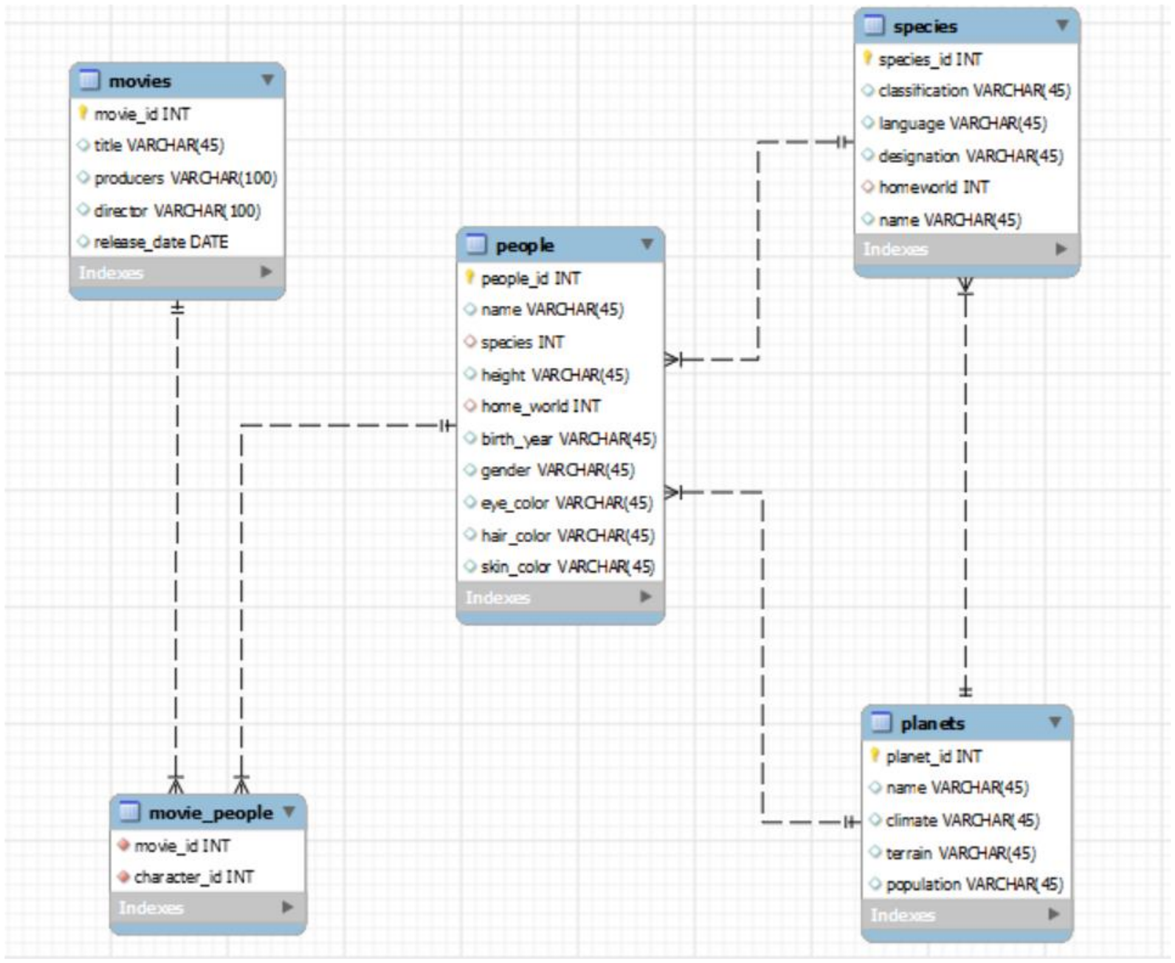
Planets table

planet_id	name	climate	terrain	population

Species Table

species_id	classification	language	designation	home_world	name

ER Diagram


movies
movie_id INT
title VARCHAR(45)
producers VARCHAR(100)
director VARCHAR(100)
release_date DATE
Indexes
people
people_id INT
name VARCHAR(45)
species INT
height VARCHAR(45)
home_world INT
birth_year VARCHAR(45)
gender VARCHAR(45)
eye_color VARCHAR(45)
hair_color VARCHAR(45)
skin_color VARCHAR(45)
Indexes
species
species_id INT
classification VARCHAR(45)
language VARCHAR(45)
designation VARCHAR(45)
homeworld INT
name VARCHAR(45)
Indexes
movie_people
movie_id INT
character_id INT
Indexes
planets
planet_id INT
name VARCHAR(45)
climate VARCHAR(45)
terrain VARCHAR(45)
population VARCHAR(45)
Indexes

8. Refactoring Strategies

8.1. Composing Methods (Extract Method):

Every method in TDG’s was doing a database connection again and again, so we extracted that part and made a separate method called “connect” which is then used instead of writing the connect code again and again.

8.2. Simplifying Method calls (Parameterize methods)

The method findCharacterDetails(..) and findCharacterName(..) were doing same task but returning different types of values. So, we removed findCharacterName(..) and utilised findCharacterDetails(..), then used the corresponding getters to get the name from the People object returned.

8.3. Generalization (Extract Superclass)

Since all the TDG classes were having username, password common. So, we created a base class called TableDataGateway and added the common functionality to it. Rest all TDG classes are derived from it.

8.4. Generalization (Pull up method)

Since all the TDG classes were having connect method common. So we pulled it up to the base class.

8.5. Generalization (Pull up constructor body)

Since the data members have been moved as in above point. So, all the TDGs pass the values to super’s constructor.

9. Coding Standards as per Java

Code conventions followed:

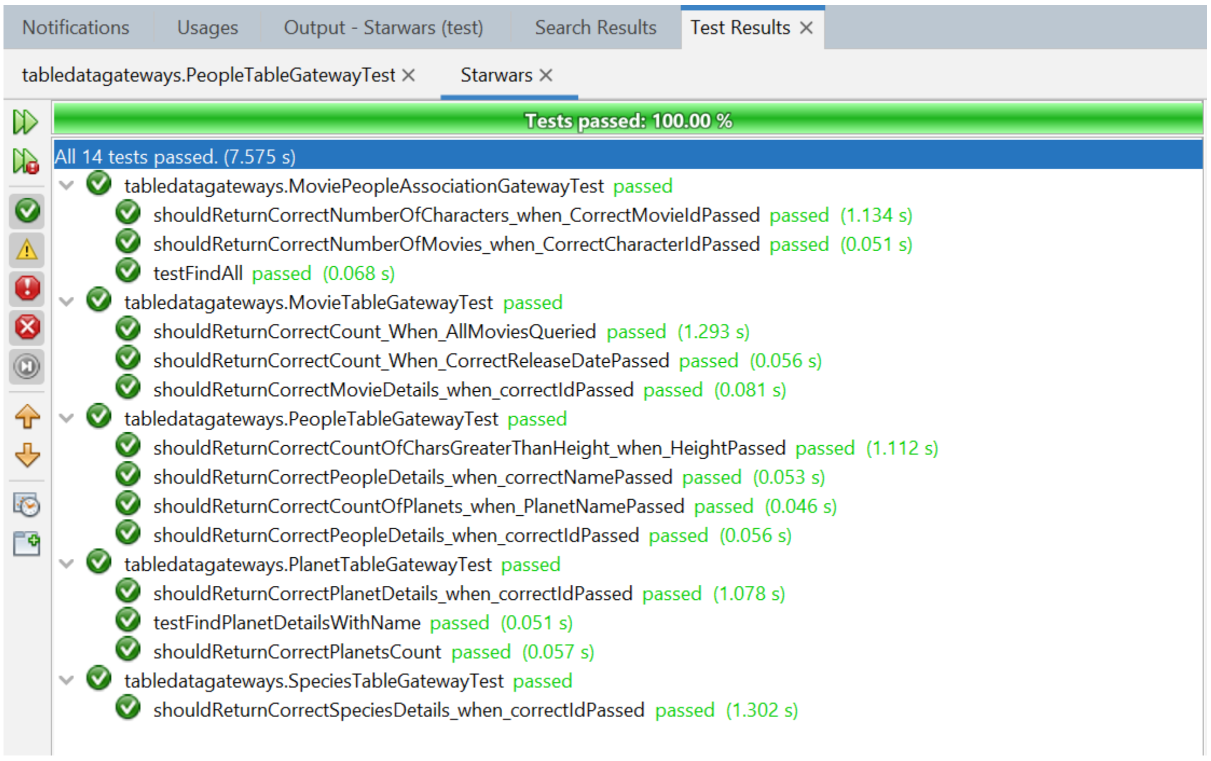
1. All the class names start with a Capital letter.
2. Package names start with a lowercase letter.
3. All the local variables or instances start with small case letters and follow a camel case convention.
4. Every class contains a header comment explaining in brief the functionality of each class and author details.
5. Code is commented wherever it is needed. Every method has a comment which explains the motivation briefly.
6. All variables , functions, identifiers have descriptive names.
7. Appropriate error handling is done to avoid unusual results.
8. Code is written in a particular format and indentation is followed to enhance readability.

10. Testing Technique used

<u>Framework used: JUnit</u>

We have used JUnit to test the application. We have created separate unit tests for each method of Java class which does some functionality. There is a total of 14 unit tests in this project and all of them pass.

Below are the test results:



```
All 14 tests passed. (7.575 s)
  tabledatagateways.MoviePeopleAssociationGatewayTest passed
    shouldReturnCorrectNumberOfCharacters_when_CorrectMovieIdPassed passed (1.134 s)
    shouldReturnCorrectNumberOfMovies_when_CorrectCharacterIdPassed passed (0.051 s)
    testFindAll passed (0.068 s)
  tabledatagateways.MovieTableGatewayTest passed
    shouldReturnCorrectCount_When_AllMoviesQueried passed (1.293 s)
    shouldReturnCorrectCount_When_CorrectReleaseDatePassed passed (0.056 s)
    shouldReturnCorrectMovieDetails_when_correctIdPassed passed (0.081 s)
  tabledatagateways.PeopleTableGatewayTest passed
    shouldReturnCorrectCountOfCharsGreaterThanHeight_when_HeightPassed passed (1.112 s)
    shouldReturnCorrectPeopleDetails_when_correctNamePassed passed (0.053 s)
    shouldReturnCorrectCountOfPlanets_when_PlanetNamePassed passed (0.046 s)
    shouldReturnCorrectPeopleDetails_when_correctIdPassed passed (0.056 s)
  tabledatagateways.PlanetTableGatewayTest passed
    shouldReturnCorrectPlanetDetails_when_correctIdPassed passed (1.078 s)
    testFindPlanetDetailsWithName passed (0.051 s)
    shouldReturnCorrectPlanetsCount passed (0.057 s)
  tabledatagateways.SpeciesTableGatewayTest passed
    shouldReturnCorrectSpeciesDetails_when_correctIdPassed passed (1.302 s)
```

11. References

GitHub link for the project

https://github.com/SonamChugh13/Starwars