LAB DP: USING THE SINGLETON AND ADAPTER DESIGN PATTERNS

1 Introduction

This lab gives you the opportunity to implement the singleton and adapter design patterns. In addition, the lab exposes you to two features of Java that we don't study elsewhere in the course:

1. the `enum` keyword, which enables you to define a new data type that enumerates several named options
2. reading data from a file using `BufferedReader`, `FileReader`, and `readLine()`.

These two features will not be tested in exams or other homework, so you should be able to make use of them in this lab based on the provided example code without having to do further background reading.

2 Getting started

1. Create your GitHub repository and import the starter code into Eclipse as described on the "How to…" page, available from the course Moodle page.
2. Unlike the other labs in this course, the JUnit tests in this lab employ JUnit 5. Therefore, remove any compilation errors in the starter code by following the instructions for "Adding the JUnit4 library to a project in Eclipse" from the course "How to…" webpage—but add JUnit 5 rather than JUnit 4.

3 The assignment

3.1 Preliminaries

Familiarize yourself with the starter code:

- Draw a class diagram showing the relationships between classes. This diagram is for your own use and need not be submitted for grading.
- Run the `main` method in `ArtistData.java` and inspect the code to understand how it works.
- Run all the JUnit tests and verify that there are no errors or failures.
- Now read the code in every Java file.

Open the two data files (data/bands.csv and data/singers.csv) to make sure you understand how they represent their data. These files store data in a very simple format known as *comma-separated values* (CSV). Make sure to view these files in Eclipse or another text editor to see the raw text format, and also open them in a spreadsheet program such as Excel. If desired, you can read more about this format on the Wikipedia page for CSV files. When you edit these files in Task A below, feel free to use Eclipse, another text editor, or a spreadsheet program. To keep things simple, however, do not allow any commas in the names of your bands or albums. As a preliminary task to ensure the code and data files are working as expected, do the following:

> **Task A:** Add the data for at least one band to the file data/bands.csv. Optionally, add some singers to data/singers.csv. Run the `main` method in `ArtistData.java` again, and verify that the new bands and singers are printed out correctly.

3.2 Creating a singleton

Note that the `ArtistData` constructor reads data from files on the computer's permanent storage. Although the files provided with this lab are very small, these files could be extremely large in a realistic application. It would be inefficient to re-read the files every time a new `ArtistData` object is constructed, as in the `ArtistAnalyzer` and `MusicStore` classes. Therefore, we can improve the design by making `ArtistData` into a singleton, using the singleton design pattern. This motivates the first coding task in the lab assignment:

> **Task B:** Make all necessary alterations to change the `ArtistData` class into a singleton. Hints:
>
> - This should require changes to four Java files.
> - Check that all JUnit tests still pass after your changes have been made.

3.3 Creating an adapter

Note that the `ArtistData` class is currently able to read data only for artists of type Band or `Singer`. We now imagine that the code is being extended to also store data about classical composers. You are going to implement this extension using the adapter design pattern. We imagine a scenario that imposes some strict requirements on how the extension can be implemented. Firstly, the current online music store that supports only bands and singers is partnering with an existing online music store that supports only classical music by classical composers. The classical music store already has code for working with classical composers, and this code will continue to be used in the classical music store. **Therefore, the code in the file `Composer.java` may not be altered in any way.** Instead, you must use the adapter pattern to create a suitable new data type that can be used with the existing classes such as `ArtistData`, `MusicStore`, and `ArtistAnalyzer`. This motivates the final task in the lab assignment:

> **Task C:** (i) Using the adapter pattern, implement a new data type for classical composers. Keep in mind that you may not alter `Composer.java`. (ii) Create JUnit tests for your new class. (iii) Alter `ArtistData.java` so that in addition to the bands and singers files, it reads the composers file `data/composers.csv`. Run the main method in `ArtistData` and ensure that the composer data is printed out as expected. Also check that all JUnit tests still pass.

4 Submitting your solution

As usual, push your code (and data files) to GitHub regularly for backup purposes and push your final version to submit the assignment.

5 Additional remark

The code in this lab is a good introduction to some very elementary techniques for reading data from files and manipulating that data. However, it is important to realize that professional-quality code for managing data would usually employ a rather different approach. In particular, the raw information would probably be stored in a database rather than in the comma-separated value (CSV) files provided here. If CSV files *were* used for production code, it would employ a well-tested pre-existing library for reading and writing the CSV files. The simple code provided here for reading the CSV files is not at all robust. Please see the Wikipedia page on CSV files for some ideas on why the approach needs to be more sophisticated.