Presented to the **Software Technology Department**
De La Salle University - Manila
2nd Term, A.Y. **2018-2019**

In partial fulfillment
of the course
In **Software Design Patterns** (**S17**)

Design Challenge 2: Observations and Explanations

Submitted by:
Ing, Andrew
Lim, Johanna
Ngo, Carlos
Villanueva, Raeanne

Submitted to:
Prof. Jordan Deja

March 15, 2019

SOLID Principles Implemented by the Project

Single Responsibility Principle

The View classes are in charge of displaying data to the user, getting the inputs from the user, and informing the user of the changes. The model classes only have to worry about storing information, and the controller does not need to worry about displaying information as it is only passing information from the Model and DAO classes to the View classes. There are also several DAO classes and Controller classes with each having to handle specific processes. For instance, the SongController and the SongDAO are in charge of managing the song objects and their data respectively. On the other hand, the UserDAO and the AccountController are in charge of handling the user's account and the customizations done by the user.

Open/Closed Principle

The current code is closed for modifications. However, it can be easily expanded because it applies the factory and the MCV design patterns. Example, if a better way of writing the SongDAO class is made, the code can simply adapt to the new one given that the new code implements the DataAccessObject interface. Moreover, if another entity was introduced in the database, the project is open to extension by simply creating a new ConcreteFactory and ConcreteProduct for the new entity.

Liskov Substitution Principle

The Song object is a stand-alone object. Nevertheless, it can be used to compose a Playlist object. There is no problem for a Playlist to add or remove a Song object from its list of Song objects as long as the user does not try to remove a Song from an empty Song list.

Interface Segregation Principle

There are no unnecessary extensions or implementations of interfaces in this project.

Dependency Inversion Principle

Because of the factory design pattern used, the classes are not heavily dependent upon each other.

Design Patterns Implemented by the Project

Factory Pattern

The relationship between classes in the DAO package is an example of the Factory design pattern. The DAOFactory class acts as the AbstractFactory that creates and sends out AbstractProducts, which in this case are DataAccessObjects. The abstract method is createDAO, which implementation varies for each ConcreteFactory. An example of such is the UserDAOFactory. It creates a ConcreteProduct UserDAO; the data access object that handles the querying and updating of data about the user in the database. If another entity was introduced in the database, the project is open to extension by simply creating a new ConcreteFactory and ConcreteProduct for the new entity.

MCV Pattern

This project implements the Model-Controller-View design pattern. The Controller classes have instances of Model classes and View classes, and they act as the "brain" of the program. The Model and View classes do not communicate with each other, thus removing any dependencies between them. If the View classes require the data from the Model classes, they should ask the Controller classes for the primitive building blocks of the Model classes. For example, instead of asking for an Album instance (which causes dependency issues), the View classes will instead ask for the album's name and cover image. This allows for faster development of the software, as the View classes can be coded separately without the knowledge of Model classes, and vice versa.

Facade Pattern

The MainController class of this project acts as the facade of all the packages. It glues together all the factories for the Data Access Objects in the DAO package, the dashboard which contains all the important panels, all of the data about the user, and the other controllers for the said data. It acts as a spokesperson for each of its children. If the View panels want to retrieve the data to be displayed, they must speak with the MainController class. If the data needs to be saved or loaded from the database, the user must speak with the MainController class. If the Controller classes want to communicate with each other, they must speak with the MainController class.

Singleton Pattern

The project implements the Singleton pattern for the database. Having many connections per client is expensive, thus, having one connection per client is preferrable, which is stored in the Singleton class. With the use of the pattern, it ensures that each client only uses one connection to communicate with the database.

DAO Pattern

The project implements the DAO pattern as seen in the DAO package. Doing this allows the software to isolate the application or business layer from the persistence layer, which in the project is a MySQL relational database. The DAO classes hide from the application all the complexities involved in performing CRUD operations in the underlying storage mechanism. This permits both layers to evolve separately without knowing anything about each other. The project contains many implementations of the DAO, specifically the UserDAO, SongDAO, AlbumDAO, PlaylistDAO, GenreDAO, and PlaylistSongDAO. These DAO classes enable the client to perform queries such as getting the list of songs, genres, albums, etc., as well as updates such as creating, deleting, and updating records.

Template Pattern

This project implements the template pattern for the hashing of the passwords. This allows the application to avoid storing passwords in plain text into the database; thus, adding a layer of security. Template pattern was used because there are primitive functions (e.g. creating salt & pepper for the hash) that should not be visible to the client.