Design Document 2 & 3

Controllers:

1.

Abstract Factory Design Pattern: When moving to Android, most of the commands turns out to be involved with button click and become one instance of View.OnClickListener class. So following Phase 1, we continue to use the abstract factory method, but refactor to fit into Android context. We have an abstract factory (SimpleButtonCommandCreator) and let all the concrete creators (all classes under controller.commandCreator package) implement the factory interface. All these creators return View.OnclickListener, i.e. all the classes in packages end with Command.

Explanation: This design is in use because we have lots of different commands on track, and that we want to segregate the implementation of the command with the Pages. Since in Android, most responses are from buttons, which should be activated by View.OnClickListener, so we decided to extend our Phase 1 program and re-established the code based on the new context using the same design ideas.

2.

Adapter Design Pattern: For the similar reason as Command, when moving to Android, the “Pages” we used in Phase 1 has to extend AppCompatActivity class and design relevant xml files so as to package as an apk and display on the phone screen. However, we have an abstract PageController that serves as a page template. To preserve our code as much as possible, we came up with a workaround, i.e. to make an abstract class PageActivity that extends AppCompatActivity with a few accommodated abstract methods similar as before. For each specific Page (almost all the class in artemifyMusicStudio that ends with Page, like the RegularUserHomePage, LoginPage, etc.), it realized the concrete abstract methods defined above and call them in onCreate() method to setup the buttons. In this case, we create an Adapter for AppCompatActivity.

Explanation: The primary reason to use Adapter is because we want to preserve Phase 1 code and project structure as much as possible. After learning Android development a little bit, there are more elegant ways to build the functionalities of each page with more flexibility, but due to time constraint, and such “refactoring” is almost rewriting the whole program, we use an Adapter to minimize our work in moving to Android.

3.

Dependency Injection Design Pattern: When implementing the concrete command classes that execute detailed requests that are sent from the user, we need to use methods from Language Presenters, Use Case classes and some other information like targetUserID, targetSongID or targetPlaylistID. However, the concrete command classes should not know the details of how to construct these objects or create data that is stored in these objects. In fact, the concrete command classes only need to invoke methods from these objects to satisfy their needs. Hence, we create a class called ActivityServiceCache that contains all necessary objects and data. Then we inject this ActivityServiceCache into every concrete command class. As a consequence, the concrete command classes are able to query the object/data that they need from the ActivityServiceCache without depending on them.

Explanation:One reason to use the dependency injection has been described above. The command classes, as “users” for language presenters and Use Case classes etc, do not need to know the construction of these objects or data. Namely, the command class should not depend on these objects. Therefore, we inject these objects to the command classes. Moreover, dependency injection provides a workaround to replace the usage of a Singleton class to preserve the same set of data that will be shared with the entire program. Since a Singleton is almost the same as an global static variable, it is unsafe to use within our program. This is because every part of the program can access and modify or even delete the Singleton class easily. It will create potential errors that it’s hard to detect.

4.

Iterator Design Pattern: For the ViewQueuePage and PlaylistSongsDisplayPage controller, we implemented the Iterator design pattern to iterate over an ArrayList and populate the corresponding Android pages with the respective elements. Specifically, in both controllers we have ArrayLists of ids of songs. We use a hidden iterator by using a for-each loop to iterate through each song ID from the ArrayList. For each iteration, we fetch useful information about the song such as its name and artist and then populate respective Android pages such that they display each of the songs in the ArrayList.

Explanation: The primary reason to use the Iterator design pattern is since it gives us the ability to iterate through each of the songs in a given playlist or the upcoming songs in the queue, which is necessary in order to display individual songs on the pages of our Android app. Not having this design pattern would limit our ability in what we would be able to display on the pages of our GUI to just ids of the songs or a very lengthy and inefficient implementation to display individual songs. Hence, the Iterator design pattern helps us implement the necessary functionality by writing code that is efficient, concise and follows the SOLID principles.

5.

Observer Design Pattern:We utilize an Observer pattern to implement our Follow/Unfollow user button and Like/Unlike button. If a user wants to follow other users, there is a Follow/Unfollow switch for him/her to send this type of request. Upon a successful execution of the Follow/Unfollow action, a user will see the word of the Follow/Unfollow button change to the corresponding Follow/Unfollow state. Similarly, if a user likes a Song or a Playlist, the heart of the Like/Unlike button turns to red. Conversely, the heart turns to white when a user unlike a Song or a Playlist. These behaviors indicate that the Follow/Unfollow and Like/Unlike button “observe” the states of whether a user Follow/Unfollow someone or Like/Unlike some Songs/Playlists. To complete the Observer Design Pattern, we implement two private helper methods to broadcast the states of the Follow/Unfollow and Like/Unlike.

Explanation: The primary reason to use the Observer design pattern for the Follow/Unfollow and Like/Unlike button is to fullfill the **Perceptible Information** principle in the Universal Design. The change of text in the Follow/UnFollow button or the turning of heart in Like/Unlike button provide precise information to the user. The user can tell directly whether his/her action is successful or not. It also satisfies the **Low Physical Effort** since now, the user can easily check whether he/she already liked this song.

6.

Builder Design Pattern: When moving to android studio. There are situations when we want to pop an dialog to either ask the user to provide some inputs or display some information that a user requests. To fulfill this type of task, we leverage the AlterDialog.Builder embedded in Android studio to build our pop up dialog. Though we do not implement this builder by ourselves, it is nonetheless a builder design pattern. To use the AlterDialog.Builder properly, we construct all necessary parts (data or information that we want to put into the dialog) and feed it to the builder. Then we invoke the builder.create() to obtain a pop dialog in our GUI.

Explanation: The reason to use the Builder Design Pattern is that we want to create the pop dialog to allow a **Simple and Intuitive Use** for the user of our app. With these pop dialog, the user can view or type in the information easily on the screen.

Entity:

In the entity classes, we initially had implemented an Iterator design pattern to iterate over the elements of an ArrayList, specifically in classes User and Playlist. However, upon receiving feedback on our Phase 1, we realized that although we were implementing a design pattern we actually made the design of our code worse by introducing a code smell. Since our entity classes were already very large due to the number of methods we had included, implementing a design pattern led to the bloaters code smell. We realized that the implementation of specific methods in the entity classes didn’t actually require us to iterate over the elements of an ArrayList as we simply needed the size of the ArrayList. Therefore, we decided to follow our Phase 1 feedback, to remove the design pattern which was taking up about 4-5 lines of code in each method and replaced it by using calling .size(), which only took about 1 line of code. Therefore, this was a design decision we made where we realized an instance of where a design pattern shouldn’t be used and refactored our code accordingly to eliminate a code smell.

Gateway:

1.

Abstract Factory Design Pattern:The Gateway classes are implemented followed by the Factory**.** This is the same as Phase 1.

Explanation: The primary reason for using the abstract factory and factory method in the Gateway class is that it provides a unify interface for other parts of the program to conduct some I/O actions like loading and saving states for ActivityServiceCache. Similar to the reason we describe in the dependency injection, other parts of the program do not need to know how a concrete gateway is implemented. Namely, we do not want the other part of the program to depend on the gateway classes. This will help us to preserve the dependency inversion principle. On the other hand, it also preserves the Open-Closed principle since now, we are very easy to include a new type of gateway to extend our I/O functionalities.

Additionally, since in phase 2 we are running the program on a virtual machine, a design decision we made was to modify our gateway output method to save the .ser files on the virtual machine rather than on our local disk.

Presenters:

1.

Factory Design Pattern: When creating the Presenter Classes, we used the Factory Design Pattern because we had one translator class that wanted to interact with various Language Presenter objects. Since our program could have many possible language extensions, we decided to use the Factory Design pattern to obscure the creation process of each of these language presenters! To be specific we have the PresenterFactory class which is an abstract class. Then we have a class called PresenterCreator which uses switch cases to create the various Language Presenter cases. Then we have our various Language Presenters (EnglishPresenter and GermanPresenter) which have their own translation methods. **As you can see, we have used the factory method to make the manufacturing of various language types easily extendable as we follow the dependency rule and uphold SOLID principles.**

Explanation: This design decision was implemented since it followed the same method as the Gateway classes. Both Gateway and Presenters are tightly related, hence staying consistent with the Factory Design Pattern will increase the readability of our program. From phase 1, we have included the translator class that is connected to the external Google api, which successfully translates text within Intellij.

2.

Adapter Design Pattern: Once we transitioned to Android Studio, we faced an issue with our phase 1 implementation, hence we had to adapt other implementations. When adapting to the Google cloud Api and Firebase Ml Library, we had to adjust our manifest and gradle scripts to accommodate the new libraries associated with each new implementation. To be specific, we looked into the **Object Adapter method** when working with the Google Cloud API as this implementation had the required methods for us to translate text for the rest of the Presenter Classes. The GoogleAPITranslator class and the FirebaseAPITranslator class are examples of the adapter classes which contain instances of the adaptee libraries and classes.

Explanation: Since our original translation method did not work with Android Studio, we had to adapt implementations from other sources. Unfortunately there was no improvement from our phase 1 as we faced further Gradle build obstacles. We used this design pattern as it was the only way to get connected to an external API efficiently while using pre-written libraries.