1. Strategy Design Pattern

Our group are creating a song recommending program which enables user to listen, get the current recommending songs, and recommend a song to other users. When we are implementing the " present the recommended songs to users" feature, we have to make a design decision: "Where do we want to implement the recommending algorithm?". Our goal is to make our program extendable for different algorithms, that is, our program could recommending songs based on highest average ratings, most listened, and most recommended by users. We want the implementation of the class to be independent of a particular implementation of an algorithm. So we decided to implement the Strategy Design Pattern. In the recommendStrategy package, we have created a IRecommender interface, inside the interface there is a getRecommend method. This enables every implementing class to use different algorithm to recommend songs. Currently, we have one class called RecommendByAvgRating that implements this interface, as its name suggests, in this class's implementation of getRecommend, it will return the recommended songs based on the songs' average rating. Then, in the usecases package, the class songManager has a method called getRecommend which is responsible for recommending songs; this method will take in a IRecommender as a parameter and directly calls getRecommend in the IRecommender interface. This way, the SongManager class don't have to know what type of IRecommender it is dealing with, all it knows is that it has a getRecommend method since it implements the IRecommender interface. In the future. If we want to have more algorithms, we simply add a new classes that implements the IRecommender interface and using different algorithm when implementing the getRecommend method. This helps our program to follow the Open Closed Principle(makes the program very extendable), and Dependency Inversion Principle(SongManager only depends on the interface IRecommender).

2. Dependency Injection&Inversion

In our program, we are saving and reading from a .ser file to implement persistence of information. For each of SongManager, NotificationCenter, UserManager which are use case classes, they all store a instance variable (a HashMap which is serializable) that contains all song/notification/user information. We want to save those variables into .ser files so we can read and save from it. However, we implemented the save and read method in the Gateway class. In order to follow clean architecture, we need to avoid use case classes directly interacting with GateWay classes. We use Dependency Inversion where we created a interface called IGateWay in the use case package, and have the GateWay class implements the IGateWay Interface. We would create the GateWay class else where in the program, then, for the constructors for SongManager, NotificationCenter, UserManager. We "inject" IGateWay as a parameter by using Dependency Injection. The advantage of doing so is that we could add more gateway classes very easily and we have avoid calling the constructors of different gateway classes that implements IGateWay Interface. For example, in the future, if we want to save and read information using a .csv file, all we have to do is create a new gateway class and make sure it implements the IGateWay interface; in the new gateway class, when it implements the save and read method, implement it in saving and reading from csv file style. Since IGateWay is being injected as a parameter into SongManager, NotificationCenter, UserManager's constructors and IGateWay is a interface,we have made these use case classes avoid direct interations with gateway classes.

3. Simple Factory Design Pattern

There are two places which we implemented the simple factory design pattern.

First one is in the GuiLanguage package inside the GUI package where we created a ILanguage Interface, and currently we have Chinese and English class implementing ILanguage; and we have LanguageFactory which contains the factory method “translateTo” which will call the corresponding constructor based on user’s choice (Chinese or English). Unlike the factory method shown in the lecture slides, instead of having a lot of if statements in the factory method, we learned from our TA’s suggestion of storing a Map<String, ILanguage> inside the LanguageFactory class to reduce the size of the “translateTo” method. Every class that implements the ILanguage interface has to implement methods that corresponds to a text message that will appear on the GUI. For example, the English implementing the ILanguage Interface needs to implement the method logout which will return the string “logout”. However, when the Chinese class implements the logout method in the ILanguage Interface it will return the string “登出” which is “logout” in Chinese. We implemented the Simple Factory Design Pattern because we wanted our program to be extendable for different languages. In the future, if we want our program to support a different language, all we have to do is create a new class that implements ILanguage and store the string representing the new class and constructor call of the new class as key-value pair into the Map<String,ILanguage> in the LanguagFactory class. Then we add a button on the first page of the GUI that enables user to choose this language. The rest of the GUI will not be affected by this change. This design will make our program very extendable as it follows the Open Closed Principle, Dependency Inversion Principle (Since LanguageFactory is interacting with the ILanguage Interface).

The Second implementation of the Simple Factory Design Pattern is no longer in the code base for phase 2 but we had it through out phase 0 to phase 1. We had this before because we were using the terminal as our GUI and we no longer need it because we have an actual GUI for phase 2. In phase 1, we created a ICommand Interface and we created different classes that implements the ICommand Interface for each of our commands(login, register....etc) . The ICommand Interface makes sure that every class that implements it will have a executeCommand button which will print prompts to the terminal for user to enter inputs. And we had a CommandFactory Class and it stores a Map<String, ICommand>, and a getCommand method which will call the corresponding constructor call for different command classes that implements ICommand based on user’s input to the terminal. This made our program very extendable which all we had to do when adding new commands to the program is to create a new class implementing ICommand and store the String - constructor call as key-value pair into the Map<String,ICommand>.