**CS345 Assignment 3 – GUI Implementation of Deadwood**

Michael Rettus, Nick Coyle

**Introduction**

Deadwood is a board game in which players pretend to be actors in budget western movies. The main mechanic of the game is dice rolling to see if acting roles have been completed successfully, and the winner has the most earnings from completing acting roles at the end of the game. We have created a version of the game with a graphical user interface using the Java Swing library. This report summarizes our use of the design patterns Model-View-Controller, Composite, Builder, Singleton, and Observer in making our game with a GUI.

**MVC**

Initially, we created a text-based implementation of the game. We used the Scanner class for getting input from the user and made calls to System.out.print to display the output on the console. The code for I/O was scattered throughout various classes. It was disorganized and difficult to change. This experiment showed us the need to somehow decouple the presentation of the data from the classes that hold the data so that could more easily change the presentation layer. The Model-View-Controller design pattern is a good pattern for solving a problem like this.

To refactor our program, we organized our java files into model, view, and controller folders. Player, Game, Room, Role, Board are the major classes that comprise the model. These classes contain the data for our program. We removed the System.out.print statements and the Scanner instance from these classes since it does not belong in the model. We also had to refactor many methods involved in taking player turns and giving the user options during their turn. We had to add a way to store the possible actions a player can make during their turn so the controller could retrieve this data. This way, the state of the player’s turn is stored in one class in the model and kept separate from the code that displays the state and the options to the user. The data is encapsulated in the models, so the controller must use getters and setters to access it.

We wrote one controller class since our program has only two views that both pertain to the game. The controller is the BoardLayersListener class. It contains a copy of the current instance of the Game. The controller has methods that get the data from the model and subsequently passes it to the view through a method call. For example, a call to the getPossibleActionsMenu() in the controller calls the getPossibleActions() method in the model.Player class which returns a HashMap<String, Boolean>. The controller method converts the possible actions to an array of String and passes it to the view. The controller is the intermediary between the data and the user interface. This means it does not matter to the view how the model stores the data, and the model does not know anything about how the view uses the data.

Our program has two view classes: BoardView and DialogView. The latter has methods to show input dialogs to the user to get the number of players playing the game and their names. It is in its own class because the BoardView class was getting too large and also because the functionality is not really part of the game board since the dialog boxes popup before the game board is populated. It makes sense to separate it. The BoardView has all the code pertaining to user interaction with the game board and the pieces on it. Here there are instances of swing buttons, dropdowns, frames, panels, labels, and everything needed for the display. There is no connection to the model. Data from the model can only be sent through the controller.

**Observer**

One challenge with the MVC pattern is maintaining consistency between the data in the model and the data that the view is displaying or gathering from the user. We used the Observer design pattern to solve this problem. Both the Game and Player models contain nested Observer interfaces. This interface maintains a list of subscribers to the model. It also has a method to add new subscribers and one to notify them when the state of the data has changed. The BoardLayersListener controller implements both of these interfaces. It subscribes the controller to the models. Anytime the data in the models changes, the models notify each subscriber—in this case the controller. When the controller is notified, it calls methods in the view to display the new data.

**Composite**

Both view classes make extensive use of the Composite design pattern. The Java Swing library has components such as JPanels and JLabels that can hold leaf objects like buttons. The composite pattern enables the components to hold other objects while at the same time having the same behavior as those objects. So the client treats both the components and the leaf objects the same.

**Singleton**

Another design pattern we used in the controller is the Singleton pattern. Since there is only one instance of the game running at any given time, there should only ever be one instance of the controller class. The Singleton pattern is a way to enforce this restriction. The controller has a private constructor so it cannot be called. The class has a private static instance of itself that is instantiated using eager initialization. We use eager initialization because we know we will need an instance of this singleton each time the program starts. Plus, it is simpler than the alternative which is not thread safe without using locks. The getInstance method returns the instance of the class.

**Builder**

We also used the Builder design pattern to parse the XML files containing the board and cards. We make an instance of the DocumentBuilderFactory and call the newDocumentBuilder method on that instance which returns an instance of DocumentBuilder. Then, we can use that instance to parse the XML files by calling the parse method.