Group Members: Sage Cook, Suhyun Ban  
CSCI345 – Spring 2022  
14 May 2022  
Assignment 2 Design Report

**DeadWood Design Report**

**Summary:**

Our implementation of the DeadWood board game features a main, singleton class called “DeadWood” which contains a Board object and several Players. The Board is responsible for interfacing with our XMLParser class to build the Room, SceneCard, and Role objects which make up the game board. Room has two child classes to represent the Casting Office and the Scene Rooms, and each overrides the Room.action() method with the appropriate behavior a player can accomplish in that type of room. Each SceneRoom object has a SceneCard, and both SceneRooms and SceneCards possess Roles, to represent the on-card and off-card Roles. SceneCards can be swapped out for new ones from the “deck” via the Board class, which keeps track of all unused SceneCards. Player objects keep track of their Room, and their Role. Throughout the game, Player objects are called upon in rotation by DeadWood to do their turn, and during their turn they can perform several actions which are either self-contained or interface with the room they are currently in. All objects with physical positions on the game board keep track of their coordinates via an ObjCoord object, although this is currently unused in the text-based implementation.

**Rationale:**

We decided to create DeadWood as a singleton class since we do not intend for multiple instances of DeadWood to be created during a single execution of the program, however we decided not to make Board a singleton as it has a one-to-one aggregation with DeadWood which ensures there will only be as many Board objects as there are DeadWood. We separated the main game logic into DeadWood and Board in the first place so that DeadWood can handle the actual behavior of the game while Board manages the physical state of the game board.

It seemed natural to make Player, Room, SceneCard, and Role their own classes since each of these has multiple instances within the game and unique properties. We gave Player two-way association with Room because Players always keep track of the room they are in, and while SceneRoom is the only type of room which actively tracks players with Roles in the scene, all rooms associate with the Player object when a Player performs an action in the room.

We created the SceneRoom and CastingOffice child classes of Room because these types of room extend the features of a basic room. The trailer room does not get its own child class because it does not have any added functionality over the basic room. The benefit of this design choice is that the Player class does not have to keep track of what kind of room it is in, and can call the same Room.action() method to interact with the room no matter what type it is.

We separated all XML parsing tasks into their own class called XMLParser because we felt this would better uphold the standard of Model-View-Controller.

**Cohesion:**

Separating XML parsing into its own class gives the XMLParser procedural cohesion since the two XML parsing tasks are not dependent on each other, but they follow a similar procedure. However, this improves the cohesion of other classes since XML parsing should not be part of their responsibility.

Player, SceneCard, Role, and ObjCoord are all designed with functional cohesion since they each only contain methods and data which pertain to themselves. Room and its child classes have sequential cohesion due to the existence of the action() method, since it is more for the usage of Player than Room, and it is only in the Room class for convenient access to Room’s data. This design improves encapsulation for the Room class at the cost of slightly weaker cohesion.

DeadWood has accidental cohesion due to the inclusion of a global Random object for dice rolls and a global Scanner object for reading user input. We created these because we felt it would be more efficient to have a single one of each than to be constantly creating new Random or Scanner objects as needed, and we did not put these in a separate class because we felt this was not enough content to warrant creating a new class. Security is also not a concern for these objects since they are for utility only and hold no sensitive data.

Board has communicational cohesion because its methods deal solely with keeping track of and manipulating data like the deck of SceneCards and the number of scenes left on the board. We designed it this way because the only responsibility of Board is to manage its individual pieces.

**Coupling:**

For the most part our project is purely message or data coupling with some instances of stamp coupling such as when Player passes itself as a parameter to Room.action(). We made this design choice since the Room needs to know which player is calling action() in order to perform the action, and all of Players attributes are private so this does not expose any data.

We have some instances of external coupling by parsing from XML files and importing from the java standard library. We also have some type-use coupling between Room and its child classes. We have one instance of common coupling which is the global Scanner object in DeadWood. We decided to use this approach despite the common coupling for the same reasons as listed above in the cohesion section.