CSCI 345: Cohesion/Coupling Write-up

Noah Duggan Erickson, Daniel Wertz

09 May 2023

The board game Deadwood is a simulation of the movie industry in the 1920s, where players compete to earn money and fame by acting in various genres of films. The game consists of a board with different locations, cards that represent actors, roles and films, and dice that determine the outcome of acting attempts. To implement this game as a program, we used object-oriented design principles to model the main components of the game and their interactions. In this paper, we describe our design choices and how they reflect the concepts of cohesion and coupling.

The primary class, `Deadwood`, is logically cohesive, handling the temporal initialization of the board and players, procedural flow of days and turn order, and temporal end-of-game logic. Thus, this class is data coupled to `Player` for the initialization procedure and message coupled to the `ViewHandler` class discussed later for handling user I/O. The high cohesion of this class is based on the design choice to have our `main()` method directly control the flow of the overall program.

The `ViewHandler` class is a top-level singleton class that is functionally cohesive single focused on interacting with the users of the program. The creation of this class was primarily motivated by a desire to create some form of MVC separation for extending into a GUI in assignment 3. The methods in this class are intentionally left vague so that the caller is responsible for the controller portion of MVC.

The `Player` class is what the user interacts with most of the time. This communicationally and procedurally cohesive class primarily handles the execution of a player's turn and assignment of roles to the player. As a result, this class couples to many classes with data coupling to `ViewHandler, Board, Office, Upgrade, Role, Set`, as well as message coupling to the `Area` interface.

`Role` is a class that effectively acts as a container for data pertaining to the currently active roles, both on-card and off-card. This logically cohesive class is responsible for providing human-readable interpretations of role data, verifying that a player is qualified to take a role, and tracking the participation of a player in a role. Therefore, this class is data coupled to `Player`. We opted to use the same class to represent roles that are both on and off-card so that there is no difference in the signatures of the roles, since they are parsed from file in the same way. However, it may be optimal to convert this class to abstract, with concrete implementations for on-card roles and off-card roles, with the payout logic refactored into each class.

Spaces on the board are represented by classes that implement the `Area` interface. These classes are `Set`, and singletons `Trailer` and `Office`. This interface enables different areas to easily be compared for checking move legality and creating a human-readable representation of the `Area`. The `Set` class is communicationally cohesive, since it pushes updates to the players about changes to the scene on that set and handles the payout logic when a scene wraps. Therefore, the class has data coupling to `Role, Board, and Player`.

The singleton `Trailer` class acts as an area that has no actions to it or any other special properties. `Office`, however, is a functionally cohesive area where players that are on the office are given the opportunity to increase the rank of their player . `Office` is therefore data coupled to `Upgrade` to acquire information about the upgrades, and to `Player` to determine if a specified upgrade can be applied. However, as of current, the `Office` class handles the XML parsing of the `Upgrade` objects, which should instead be handed off to the `BoardParser` class.

Cards are represented by the `Scene` class, which are a container for data about the scene alongside the `Role`s for that scene. Therefore, this class is functionally cohesive, since its methods are solely focused on representing that scene.

The game board is represented by the singleton `Board` class, which at the beginning of execution runs the parsers on the data files and stores the cards and areas internally and handles the "reset" of the board that occurs at the end of each day. Therefore, this class is communicationally cohesive, as it makes updates to `Scene` objects by placing them on `Sets`.

The XML parsing section of the program consists of three classes: an abstract `Parser` class and its extensions `CardParser` and `BoardParser`. In the current implementation, the `Parser` is only coincidentally cohesive, containing methods that are common between the two parsers. In future, this could be improved by restructuring the methods in the extensions to allow for the creation of more generic methods in `Parser`. Meanwhile, `CardParser` and `BoardParser` are sequentially cohesive, and coupled to `Scene` and `Set/Area` respectively.