**Sorcery – Design and Documentation**

CS 246 – Assignment 5

December 4, 2017

Catalin Floca-Maxim, Jafer Haider, and Berges Irani

**Introduction**

**Overview**

Sorcery begins by taking the players’ names and the location of their respective deck in the main function. Main then constructs the BoardController, or the “game loop” responsible for the underlying logic. BoardController first constructs the required players’ data stored inside the BoardModel, as well as establishing a display to the user (either a text or graphics display, or both). This is a basic MVC foundation upon which the rest of the game is built: the BoardController represents the “controller”, handling input, and manipulating data (the “model”) which is transmitted into the “view,” TextDisplay and GraphicsDisplay. Transmission of data into the view is handled using a modified Subject-Observer pattern, where the BoardController acts as a subject (but it does not inherit an abstract subject [EXPLAIN WHY]), and TextDisplay and GraphicsDisplay inherit an abstract observer.

Upon initializing the data, the BoardModel stores the data of the two players on its stack. The players’ data consists of the Player class, which contains fields for the players’ health, magic, and pointers to their respective cards. The physical cards are always allocated on the heap. This is because cards move constantly throughout the game: from a hand, to the field, to the graveyard, and potentially even to the other player. In addition, cards often need direct access to other cards to modify other cards’ data, as per the game’s features. Allocating the cards on the stack would potentially make movement and access of the cards more difficult, especially when a card must move to the other player. As a result, the cards are kept on the heap, with smart pointers accessing their data located on the Player’s stack. Shared pointers to cards are always used in lieu of unique pointers, because it is more efficient in moving cards: where a unique pointer would require a deep-copy move assignment when a card moves from the deck to the hand, a shared pointer simply requires that we create a new pointer to the same card, and remove the previous pointer.

The cards themselves are organized in a top-down hierarchy starting with an abstract class, *Card*. Card allows every subclass access to the BoardModel through a static pointer to BoardModel. This provides every card the access it needs to any card and player in the game, which allows for unlimited card customizability (specifically spells, rituals, and enchantments) and a great deal of flexibility in adding more features in the future. Furthermore, Card has a pure virtual *updateState* function that concrete subclasses must implement. Thus, the rules of APNAP order can be easily implemented through this function, which takes in a list of the events that occured (by reference), so that every card can act accordingly.

Player directly inherits Card. In doing so, Player implements *updateState*, which simply calls that same function for all of the required cards (i.e. the ritual and the minions on the field), passing forward the list of events that occured. Again, this allows implementation of APNAP order to be done in a very simple and organized fashion. Furthermore, the different types of cards inherit from Card through a diredct subclass of Card, called *NonPlayer*. NonPlayer provides public getters and setters for for data such as the magic cost of a card, or the card’s current owner. It also provides the *cast* functionality, which is the main way cards are utilized in the game. As such, every physical card implements this function which can be customized according to the respective card.

NonPlayer then breaks up into each type of card: Minion, Spell, Ritual, Enchantment, and also Ability, to represent triggered and active abilities (although Ability and its derived classes are never used as physical cards placed on a board, rather as an attachment to a minion). The Minion and Ability class follows a modified Decorator pattern [IS THIS TRUE?], where Minion acts as a concrete component and Ability is a decorator. However, Minion *owns* Ability, since a minion’s ability moves wherever the minion moves throughout the game. Moreover, only Minion is a concrete class, since all Minions do the same thing: they attack, and the *cast* function simply uses an activated ability. Thus, the only features differentiating minions are their names, attack, defense, and the abilities they own. Spell, Ritual, and Enchantment are abstract, and the physical cards corresponding to each type inherit directly from these classes.

**UML**

**Design**

**Resilience to Change**

**Questions**

**Extra Features**

**Final Questions**

**Conclusion**