Sorcery TTK

By An (Tony) Zhang, Qing (Tim) Liu, Kevin Wen

1 Introduction

The game of Sorcery TTK is a card game based on collectible card games such as “Hearthstone: Heroes of Warcraft” and “Magic: the Gathering”. In the game, there are two players and aiming toward the same goal of eliminating the opponent (reducing the opposing player’s health to 0. They will be competing using their own deck of four different types of cards. The four types of cards, Minions, Spell, Enchantment and Ritual each has unique special abilities, whether dealing damage, buffing, or debuffing etc. Each player would take a turn to play their cards and perform action aiming toward the goal of eliminating the opposing player while protecting himself or herself. In this project, we used multiple design patterns, which will be explained in detail in the report, to achieve the goal of OOP design -- low coupling and high cohesion.

2 Overview

Sorcery is an interesting game itself that attracts us from the very beginning of the project. One of the reasons is that we chose the Sorcery project is that although Sorcery may look like a complicated program to implement. However, the overall structure of the program is easy to be determined because the game only has two players and four collections. Here is an overview of the structure of our program:

Start from our most basic class **Card**, as cards are basics of Sorcery. During the gameplay, the change of state of a card may impact all other cards in the collections. Therefore, We used the **observer pattern** to implement the basic gameplay of Sorcery to maximize the code reuse. Since the cards could impact other cards, cards are both subjects and observers.

There are four types of cards, **Spell**, **Minion**, **Ritual** and **Enchantments**. The card class has fields that all cards possess such as the cost, the description, the name… etc. For convenience, we define all functions needed for each card whether it is a minion, a spell, a ritual or an enchantment in the card class as virtual. And later, when we are implementing the specific type classes, we then implement the needed functions of that type of card. For example, we define get_defense() as a function of Card class, and we only implement it in the Minion Class as only minions have defence value.

We decided to create a unique class for each specific card of the game as this would make our program more **resilient to changes**. For each new card, we only need to create a new class and add it to our existing card list. Another reason for creating a unique class for each specific card is that it follows the **single responsibility principle**, as each class we created only responsible for one specific card.

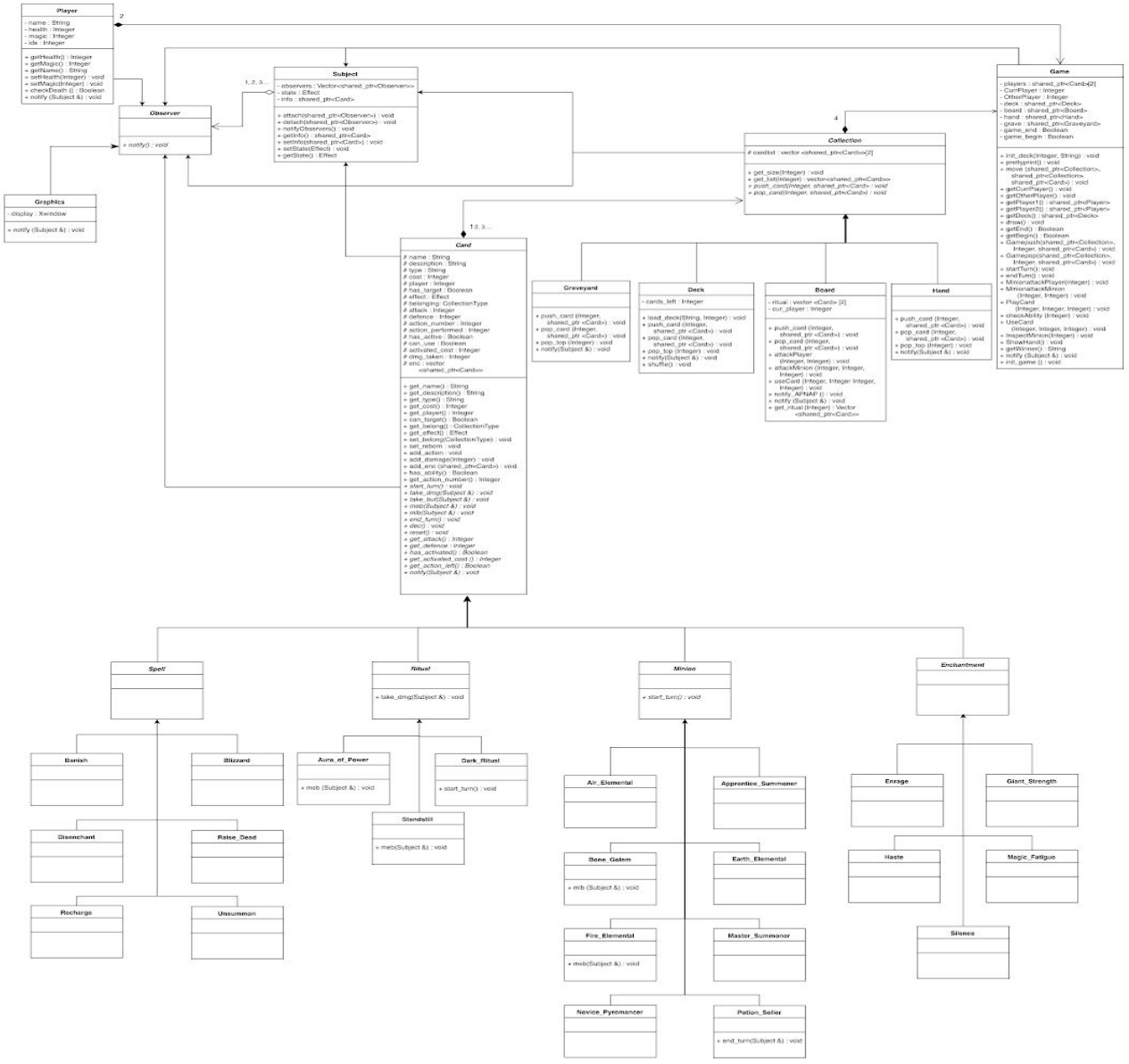
Next, the four types of collections, **Hand**, **Deck**, **Board** and **Graveyard**. During the gameplay, the four collections are where the “battle” takes place. Since those four features have similarities -- they are all a collection of cards. Therefore, we implemented the abstract class Collection contains two card vectors storing cards, one for each player. For each of Hand, Deck, Board, and Graveyard, we

implement their own specific features used for the gameplay. For example, the Deck would have a field indicating the number of cards left, the Graveyard would have a function returning the top card in it for the use of Spell Unsummon… etc.

Since the game only has two players and reads the input from the command line, we adopted the **Model-View-Controller** design pattern where the game class wraps everything in the game including the two players and four collections. After each command is inputted, the main function will read the command and perform specific actions on the game.

In a nutshell, Sorcery TTK is a well-designed card game that achieves the main goal of OOP design and project requirements.

3 Updated UML



4 Design

4.1 Design Pattern

We applied several different design patterns and strategies in our project, namely:

4.1.1 MVC Pattern

At a high level, the game follows the MVC pattern. We divided our program into 3 parts, which each is responsible for a role in MVC pattern: the main function reads in a user input (“controller”), card, collection, and game manipulate the underlying game state (“model”), and graphics display and prettyprint() in the game class display the information about the game state to the user (“view”).

- **Advantage**:

1. All three of these tasks are modularized so that any part can be switched out for a similar part that provides the same higher-level functionality. For example, the Graphics class used to output information can be replaced by a high-level GraphicsWithMouse class that support mouse actions.

2. We will not affect the entire structure and other modules if we want to do some modification or addition to the program. Since we have separated our program into different modules and modifications in one class or module will not influence other modules.

4.1.2 Observer Pattern

Since we need to keep track of the real-time changes of any action in the game, we decided to use the observer pattern to solve these “changing status”. Observer Pattern is implemented by Card, Collection (Subject) and Collection, Graphics, Card, Game (Observer). Each concrete subject has a unique notifyObserver() which depends on the features of the card. Once the observer of the subject gets called, the notify() method of it will act in a corresponding way in the response of the information they dig from the subject.

- **Advantage**:

1. It will **decrease the coupling** between objects since every class is responsible for its own field and method and don’t need to rely on other classes. The way to build connections between classes is to use notifyObserver() in subject classes and notify() in observer classes.

2. It supports **adding different observers** and doesn’t need new clients or programmers to fully understand other parts of the program. Since the observer pattern can help us add other observers independently, the clients or new programmers in the group can add other observers if are needed. It makes the project more modularized and easy to add new features.

3. It supports **adding new subjects (cards)** without knowing the implementation of other parts. It is very flexible for us to implement new cards by just creating a new concrete subject and implement its featured methods.

4.1.3 Template Method Pattern

Template Pattern is implemented in the following classes: Observer, Subject, prettyprint() in Game. Through this way, we can use this template for different observers and subjects that have different information type as their arguments.

- **Advantages**: We can add different observers freely without the limit of the type of arguments. Since different types of Card will have different functions and behaviour. We can modify the information type of different cards' arguments as needed.

4.2 Shuffle Strategy

We shuffle the deck by first draw too random integers as seeds. Then, we use these two numbers to modulus the size of the deck to get the two final random number. In this way, we limit the two final random numbers within the size of the deck. Then we swap the two cards on the position of these two final random numbers. By repeating this procedure for 100 times, we can get a totally shuffled deck for players to draw cards from.

4.3 Coupling and Coherence

The design reflects relatively low coupling and high cohesion, which is the aim of Object-Oriented Programming.

Coupling: The highest level of coupling in the design is the Game class that wrap all the components of the game and keep track of all the changes. This is the only class rely on other classes and all other concrete classes can be implemented and used independently. As a result, the coupling of the program is relatively low.

Cohesion: The biggest class in the project is the Game, which is a wrapper and container to all the components of the game, which has many methods doing different tasks. Apart from the Game, all other classes are doing their own jobs. For example, Card does the tasks of the card, Board is responsible for the tasks of board things. Thus, in general, most classes in the whole project are performing exactly their own tasks and therefore present high cohesion.

5. Resilience to Changes

As we mentioned in the Overview and Design parts, our design complies with the principle of high resilience. Hence, flexibility was reserved in situations that are likely to change in the future, in this part, we will describe in detail about the resilience of the different modules of the Sorcery TTK.

5.1 Display

Sorcery TTK supports adding different and multiple displays by adding observers to the Game class. Since Game is monitoring all the changes in the game in real-time, the displays can also support real-time changes just like "Hearthstone".

5.2 Extra Cards

Every extra card can be implemented as a concrete class of one type of the Card (Minion, Spell, Ritual, Enchantment). We only need to construct this card first by consuming its information, and then implement its special methods, all the other regular cards methods like getters and setters. This feature is benefited from the observer pattern and the design of our concrete subject. In this way, we maximize the code reuse and simplicity to add new cards.

5.3 Multiple Players

Since we implemented players as observers to the board, it is easy for us to add in additional players into the game. Afterall, Sorcery would be a very fun game if it can be played in turns with more people. By using observer pattern, we can easily attach more players and their respective decks into the game and then take turns to play the game. This would be easily achievable and require little modification of our existing code.

5.4 Multiple abilities (the combination of triggered and activated abilities)

Our implementation of card abilities involves using a state to define the core mechanics of a card and use observer patterns to interact with subjects, which is convenient and flexible, especially in a game like Sorcery where there can be multiple events happening simultaneously due to the interactions between cards. Since states are compact and stores enough information for the observers to interpret and react to them, we can easily support additional abilities of a card or minion. All we have to do is to have more states and override more basic methods for a card to easily support the additional features that bends well with the existing archetypes of behaviors.

6. Answers to Questions

6.1 How could you design activated abilities in your code to maximize code reuse?

In our design, we achieve a high level of abstraction using the Observer design pattern. We store everything crucial about a card inside it's state, which is the Effect structure in our program. The effect structure describes what the card does as it's activated ability or the behaviour for triggered ability. Not only that, the Effect structure even allows us to describe spells and rituals so we can handle them using similar functionalities in our code. This way, we can utilize the Observer pattern to its fullest. One notify function of a class can handle most general archetypes like taking damage or receiving buff easily, so we do not have to do extra work when introduce new abilities, but rather grouping them with existing genres to maximize the amount of code reused for similar types of operations.

6.2 What design pattern would be ideal for implementing enchantments? Why?

The decorator pattern would be ideal for implementing enchantments. Every time a new enchantment is applied, it can be view as a decorator being applied to an existing minion. In addition, by using a decorator pattern, it is easy to change a minion's stats and behaviour by overriding the basic methods of a minion and adding new features based on the previous enchantments. However, there is a major problem which is disenchant. We will be facing problems about de-decorating our decorators to undo the top enchantments. To address the problem, one possible approach is to store a base pointer to the

underlying minion and do some extra work when doing other works, but it is still very flexible due to the nature of decorator pattern. Unfortunately, our group did not have the time to complete the implementation of a decorator pattern and we instead adopted a vector approach to store the enchantments and interact with them through overloaded basic methods. Still, a decorator design pattern allows us to have way more control over the ability of enhancements so they can do more than the are currently designed to do.

6.3 Suppose we found a solution to the space limitations of the current user interface and wanted to allow minions to have any number and combination of activated and triggered abilities. What design patterns might help us achieve this while maximizing code reuse?

The observer pattern would again be ideal for this kind of situation. We can have the notifier going through a number of activated or triggered abilities simply through going through their respective states and activate or trigger them as required. These new functionalities can be done through overriding existing methods so that code reuse is maximized in the process. Similar to the above question, much of the existing functionality can remain unchanged. In addition, through the observer pattern, we can have multiple abilities of a minion reacting to different kinds of events when notified, hence making observer pattern the ideal solution to such situations. In our implementation, we choose the observer pattern as part of our core ideology for implementation for the same reason so our code is more resilient to change.

6.4 How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

The ideal solution is again to utilize the observer pattern. By using observer pattern, one subject can notify multiple interfaces, which then makes them exhibit expected behaviour. Since the observer pattern is utilized, very minimal changes are needed to add support to the new interfaces as we can easily attach them to desired subjects to gain necessary data from them when notified.

7. Extra Credit Feature

7.1 Smart Pointers

Smart pointers simplified memory management overall. The main challenge was in differentiating between when shared/unique pointers needed to be used, and how to work with them. As for detecting other issues, for the most part, shared pointers were manipulated in standard cases that were unlikely to cause problems. For the sorcery project, there are twenty-two concrete classes. We chose shared pointer to deal with them when they were created during the game to achieve the goal of managing the heap memory conveniently and efficiently.

8. Final Questions

8.1 What lessons did this project teach you about developing software in teams?

Teamwork is the most crucial component in a group project. Clear communication and dispatch of each group member's job are also very significant.

At first, group members need to gather together to discuss the outline and structure of the whole project. After everyone agrees on one designing solution, the group can start the true implementation.

Everyone in the group should be able to implement some parts of the project and make progress individually. Moreover, version control (git) and knowing how to use software development tools is a very important skill that is needed in a group project. The group members should keep the project updated and always commit and push his/her progress on the project. It will make the project much more efficient. During our implementation of Sorcery TTK, we tried to follow best practices and sometimes worked together if necessary.

This project also teaches us a lot about how to overcome the challenges of working on a large project with many edge cases and complicated requirements. A very effective method is to work incrementally from one working version to another working version. The agile process that is commonly applied in industrial software development is very useful in this kind of project. Choosing a suitable IDE is also very important. At the beginning of our project, we decided to use Clion and turned out to be very useful and convenient for debugging at last.

8.2 What would you have done differently if you had the chance to start over?

First, We would have started the project earlier as we sacrificed a lot of our sleeping time on the last three days before the project deadline. Time management is very important in the group project. A better plan will result in a more efficient working process.

In addition, we would do more jobs in the logic and structure parts at the beginning. During the implementation, we confronted a lot of situations that we need to modify our original structure. If we could have our structure, patterns, and logic clearer at first, it would be easier and more efficient for us to handle the real code part.

9. Conclusion

Sorcery Project is overall a challenging project for our group of three, and definitely gives us hands-on experience. Over the course of finishing the project, many skills and lessons were learned.

Firstly, as mentioned in the previous section, teamwork is important. A great program comes not from an individual's excellent work, but from a group of developers' collaboration and cooperation. During our project, we distribute our works evenly and programmed the whole project together. When a hard project was distributed into simpler smaller tasks, it is a lot easier to implement.

Secondly, planning beforehand. We started the project early by discussing our design of the program, but however, we did not create a detailed plan to plan out our week's work. This is one of the major problems we faced as we were running out of time and had to stay up late.

Last but not least, design wisely. We started the project design discussion early, but we did not think deeply into the design of the program. This left us struggling when implementing the project. Having a nice UML diagram could help organize thoughts and keep the logic and overall structure clear during implementation phase. To do that, when we are doing design before the actual coding, we should think deep and try to write out the functions and fields of each classes.

Overall, our group enjoyed this project as we are all fans of collectible card games like Hearthstone. It may give us a really hard time, but when we see it running, we all get that sense of achievement. We all learn a lot from it.