CS 246 Final Design: Sorcery

By: Amaya Ko and Diana Tatar

1.0 INTRODUCTION

Developing Sorcery within the confines of the terminal, ensuring the incorporation of various object-oriented principles and a scalable game structure for future changes posed a significant challenge for our team of two. Beyond just the fundamental user commands and elements of decks, hands, boards, and graveyards, we sought to encapsulate the intricacies of a strategic card game like Magic: The Gathering. This report delves into the structure of our implementation, highlighting the fundamental object-oriented programming concepts that shaped our approach. the challenges we encountered, and the strategies we employed to overcome them.

2.0 OVERVIEW

Our sorcery program follows a typical object-oriented structure to effectively manage the complexities and interactions within the game. It is composed of several key components, including the main game loop, input management, the game board, the players, the cards (including all the various types of cards and their abilities), and the display interfaces.

The main game loop owns a game board and processes user input to then subsequently update the board and its displays. The game board consists of two players as well as a vector of displays that are updated when the game state changes. The board manages the interactions between the two players, such as when one attacks another's minion, and notifies the game's display interfaces to reflect the changes made to the state of each player and their cards.

Players, in addition to their individual information such as their names, life points, and magic points, contain several vectors of pointers to cards which represent each player's deck, hand, graveyard, minions-in-play, and ritual-in-play respectively. Each player manages the location and movement of each of its cards. For instance, moving a minion card to the board when it is played.

Cards, specifically the different types of cards, are implemented using inheritance. Our superclass contains the basic information of every card such as its name, cost, and description. Our subclasses – Spells, Minions, Enchantments, and Rituals – manage the attributes and behaviors specific to each type of card.

By implementing our program with object-oriented principles, we were able to leverage code reusability and logically connect each component to enhance code organization and ability to collaborate. It also facilitates the addition of new features and ensures resilience to change.

3.0 DESIGN

Beginning at the highest level, we used Model-View-Controller (MVC) architecture to implement our game loop, game board, and display interfaces. The MVC architecture consists of our Sorcery, Board, and abstract Display classes. The Sorcery class acts as our controller and manages user input, including errors when handling invalid input, and gameplay, such as turn-taking. After receiving valid commands and input, our board is then called to implement the respective action/behaviour. The model, our Board class, stores and manipulates the players and other game data. Our abstract Display class acts as our view, receiving information and updates from our model and controller to display the game board to users.

Within our MVC architecture is also an observer pattern among our Board and Display classes. The Board class acts as our subject which contains a vector of concrete Displays (observers). The Displays class acts as our abstract observer class. TextDisplay and GraphicsDisplay, inheriting from Displays, are our concrete observer classes that display our game board. When a new display interface is needed, another concrete observer can easily be added to the board's vector of observers. To access the game information required to display the game, the displays contain smart pointers to the board's players. When the board's players/state is changed, each display is notified via a loop that calls on each display's notify method. When notified, the game is re-displayed to reflect the updates.

Our cards, specifically each type of card, is implemented using simple inheritance. Our abstract Card class contains the shared attributes amongst all cards – name, cost, and description. Our card subclasses inherit from the Card class and represent the different types of cards: Spell, Minion, Enchantment, and Ritual. The individual subclasses implement the specific behaviours and functionalities unique to each type of card via additional methods and fields.

The Minion-Enchantment relationship is implemented using a decorator pattern. In this case, the Minion class acts as our object whose functionality and behaviours are being changed, and the Enchantment class acts as our decorator, the class containing the changes that are applied to a minion. By applying the decorator pattern, we are able to dynamically modify the behaviours and functionalities of individual minions with a variable amount of Enchantments. More specifically, as new enchantments are added to a minion, they are applied in oldest-to-newest order, and when a minion leaves play, its enchantments are automatically removed.

Finally, for the relationship between Minion and Triggered Abilities (Ritual), we used another observer pattern. The Minion class is both a subject as well as an observer of themselves. When a minion enters the board (is played), it becomes an observer of the other minions also in play. In turn, the other minions on the board are also attached as observers of the newly-played minion. Furthermore, Triggered Abilities, which are also Rituals, is also a concrete observer class. Triggered Abilities are attached to Minions as observers. Each observer, whether it be a Minion or a Triggered Ability, is notified as needed to enact its respective actions. Specifically, Triggered Abilities has four different states – beginning of turn, end of turn, minion entering play, and minion leaving play. When the conditions of its respective state are met, the triggered ability is notified to execute its action.

4.0 RESILIENCE TO CHANGE

By design, the structure of our program easily accommodates program changes and new features due to our use of object-oriented programming concepts and design patterns such as MCV architecture, polymorphism, observer patterns and our adherence to the single responsibility principle.

Some ways in which our program could be changed is with new game commands and/or new command-line flags that add/remove specific functionalities of the game. Our program would be able to accommodate these changes due to the fact that we implemented MVC architecture for the main game loop. Our Sorcery class, the controller, is responsible for handling user input and serves as the connection between the user and the underlying program logic. Since the controller delegates the actual processing of commands to the Board class (model), incorporating new flags and commands only requires the programmer to extend the Sorcery class to recognize and interpret new input. This abstraction ensures that the board, which encapsulates the core logic and data, remains unchanged. However, if the purpose of the new flags/commands are to add new features/functionality to the game, the Board class would have to be modified to accommodate the new game logic and behaviour. This applies to new game rules as well.

The input syntax of commands could also be easily modified. As previously mentioned, the controller handles the user input and connects it with the game logic in the board. As long as the user input appropriately translates new input commands into method calls in Board, the program can easily accommodate different input syntax without necessitating modifications to any other class.

By incorporating an observer pattern between the Board and Displays classes, we've ensured the flexibility and extensibility of our program in regards to additional game display interfaces. The observer pattern establishes a dynamic relationship in which every registered display (concrete display classes) is notified when the board updates/changes state. This structure allows for easy accommodation of additional interfaces without altering the board, the game logic, or the interactions between existing components. When introducing a new display, a new concrete observer class can be created that implements the notifyDisplay method to define how the new interface responds to changes on the board. Once created, the new Display observer can be added to the vector of board observers to seamlessly integrate the new display into the existing program.

Another modification our program supports is the addition of different spells, minions, enchantments, and rituals. Due to the structure with which these different card types are implemented, creating different cards of each type is simple. Each card type class acts as a template with its behaviours already encoded into the class' methods. To add a new card only requires setting the default fields of the card in the constructors of its respective card type. The simplicity of adding new cards allows for the adaptation to evolving game dynamics and expansion of the card repertoire without compromising the integrity of the existing code.

Furthermore, not only can new cards of existing card types be added, but our program can also support the addition of different types of cards entirely. The polymorphic nature of the Card class serves as an overarching blueprint for all card types. This enables the addition of new card types effortlessly as it only requires the creation of a subclass that inherits from the Card class as well as minor additions to the display classes to display the new type of card. Each new type of card can then introduce its unique functionalities through its fields and methods. This design flexibility ensures that the program can adapt and grow with the inclusion of new card types while maintaining a cohesive and scalable structure.

A final modification we'd like to mention is the ability to make Sorcery a multiplayer game. Adhering to the single responsibility principle, each of our classes has a singular purpose. Specifically, our Players class is for storing each player's data and our Board class is to manage the interactions of its Players. By structuring our program this way, we can easily scale up the board to handle a variable number of players by creating additional instances of player objects within the Board. Furthermore, the existing commands and actions of each player can accommodate additional players by modifying the limits of valid input. However, to note, when new players are added, minor changes would have to be made to the display classes to accommodate displaying multiple players. Thus, overall, our program structure provides the foundation for multiplayer dynamics.

In summary, our use of object-oriented programming principles and design patterns foster flexibility and extensibility in our program. Through MVC architecture, polymorphism, and the observer pattern, our code can seamlessly accommodate diverse modifications and additions including, but not limited to, new game commands, altering input syntax, expanding the array of card types, and multiplayer functionalities. As well, the single responsibility principle guides our various classes, ensuring a modular structure that facilitates easy modifications without needing to alter the rest of the code. This adaptability positions our program to be resilient to change, allowing for continual growth and potential future enhancements.

5.0 ANSWERS TO QUESTIONS

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

Previously, we thought of designing Activated Abilities as an inherited class of Spells. Since Activated Abilities cost magic and an action point to use, we believed they worked similar to playing a Spell card, with the exception that Minions use activated abilities, while Players use Spells. Thus, we initially thought we could make use of NVI (non-virtual idiom) in the Spell and ActivatedAbility classes. By defining public common methods with shared logic as part of the client's interface, it would have allowed for customization between the both the Spell and ActivatedAbility classes. We believed this would've maximized code reuse. However, we did not end up implementing our code this way. Instead, we created a basic Ability class. Derived from this were the subclasses TriggeredAbility and ActivatedAbility. The two types of abilities share the fields of card name and description, and the fact that they are applied to a Card to provide some sort of ability. Implementing activated abilities in such a way improved code reuse as the two abilities shared code for the common Ability ctor, and getter and setter methods for basic fields. Additionally, the sorcery.pdf file mentions that enchantments can grant a minion a new activated, or passive ability. Enchantments could thus utilize the basic ability class to store the ability they provide, be it an activated ability, triggered ability, or none of the above. This also contributes to maximizing code reuse.

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

At the beginning of this project, we believed the decorator design pattern would be ideal for implementing enchantments. This has not changed. As previously mentioned in the design portion of our report (Section 3), we implemented the Minion-Enchantment relationship as a decorator pattern. This enabled us to dynamically add and remove functionalities/behaviours of Minions and easily add each Enchantment onto a Minion as well. The decorator pattern also supports easy removal of Enchantments associated with a Minion which allows us to easily remove a Minion's Enchantments once they leave the board. Implementing this design pattern also enabled us to create multiple concrete Enchantment classes, each with a different functionality. Thus, we were able to properly implement the various types of Enchantments described in the project specifications. Finally, with a decorator pattern, each decorator wraps around another decorator, creating a chain of modifications/functionalities. This characteristic allowed us to maintain the oldest-to-newest order in which Enchantments are applied.

5.3 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?

.

Initially, we considered both the Decorator and the Template Method Design Patterns to help us maximize code reuse and allow minions to have any number of activated and triggered abilities. We've since refined our thinking to focus on the Decorator design pattern. Specifically, we could implement an AbilityDecorator of Minion with decorators for TriggeredAbility and ActivatedAbility. This would encapsulate the decorations (abilities) and allow us to combine them in any quantity and order, providing for significant code reuse. The base Minion class would remain reusable and unchanged across different minions, with these ability decorators customizing each minion.

In terms of implementing the APNAP order of the now multiple activated and triggered abilities, we could use the Template Method to clearly define the order of steps taken when unravelling the play-order of a minion's abilities. Depending on the order in which these were stacked, that could become a very complicated process. Using this design pattern would allow us to only define this process once, and follow it for all minions, providing for additional code reuse.

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

Initially, we thought of implementing the Model-View-Controller (MVC) architecture with an observer design pattern to support multiple interfaces. Since our game doesn't require the details of its display interfaces to run, this was the ideal strategy that ensured minimal changes to the already existing code.

We had planned for the MVC architecture to consist of our Sorcery, Board, and abstract Display classes. The controller, our Sorcery class, would manage user input and gameplay (ex. Turn-taking), as well as control the flow between our model and view. The model, our Board class, would store and manipulate the game data. It would also act as the subject of our observer pattern and would contain a list of Display observers. Additionally, the view would be our abstract Displays class which would act as an observer to our Board class. Concrete observer classes, such as our TextDisplay and GraphicsDisplay classes, would be added to a Board's list of Display observers and would be notified as the board is updated.

As discussed previously, our implementation followed this statement fairly well. By implementing our game with an observer pattern, we are easily able to add more interfaces to the existing program by creating more concrete observer classes and adding them to the Board's list of observers. And by maintaining the MVC architecture and keeping our controller, model, and view classes separate, adding such an interface would only require us to modify the list of observers in the Board, requiring minimal change to the rest of our program.

6.0 FINAL QUESTIONS

6.1 What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

The main lesson we learned from this project and developing software in teams is that it is very difficult. However, we also learned various lessons and strategies that helped ease the process.

One of the main lessons learned was the importance of setting goals and expectations at the beginning of the development process. This included overall expectations of the final product as well as completion dates for each aspect/component of the game. By doing so, we ensured we were on the same page regarding priorities and quality of work. As well, sharing the goals and expectations of task completion dates also compelled us to create a work schedule which allowed us to get each of our parts completed in time.

We also learned about the importance of consistent communication. Working on such a large project in a relatively short amount of time, with each partner relying on the other to complete their part, would be impossible without consistent communication. While developing our program, we were in constant communication with each other – updating each other on progress, catching errors, and bouncing ideas off each other. The clear, open channels of communication fostered a cohesive work environment and ensured that everyone was up-to-date on the project at all times.

Another big lesson we learned was knowing how to leverage each of our different strengths and abilities. Fortunately, we have worked as a team before and already had a previous understanding of each of our unique talents. This enabled us to split up the workload such that each team member could work on what they were best at, maximizing our potential as a team and ensuring that our final product was done to the best of our abilities.

Overall, this project has been an illuminating experience. However, amid the challenges, we learned invaluable lessons and strategies that helped ease the process such as setting clear goals and expectations, consistent communication, and leveraging our diverse strengths and abilities.

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

If we had the chance to start over, the first thing we would have done differently would be to acquire a third team member. Throughout the development process, we realized the sheer magnitude of the project and the challenges that would accompany it. Both the coding itself and the time constraints on completing such code proved very difficult to get done with only two team members. If we had had a third programmer, we would've lessened the workload for everyone, which may have given us the time we needed to fully complete each component to the standard at which we wanted it to be.

Furthermore, given the fact that we were a team of two, had we the chance to start over we also would have chosen an easier game. Reflecting back on our choices, choosing the game that had the disclaimer of how heavy the workload for it would be was not the best decision. For the same reason as before, choosing a project with a lighter workload would have allowed us to complete our game at a higher quality.

We also would have started coding earlier. Despite making and relatively following a comprehensive work schedule with set completion dates, it was still quite a time crunch trying to complete everything. Had we had the chance to start over, we would have attempted to complete our initial plan and UML (due date 1) earlier, allowing us to start on our code earlier. This would have given us a full two weeks to write our program versus the week and a half that we gave ourselves initially.

In conclusion, there were various aspects of the project that we would have approached differently if we were given the chance to start over. This ranged from the composition of our team, to our choice of game, to the timeline of the project itself. Despite this, we were still able to complete the project to a relatively acceptable standard and are proud of our final product given our circumstances.