Sorcery – Design and Documentation

CS 246 – Assignment 5
December 4, 2017
Catalin Floca-Maxim, Jafer Haider, and Berges Irani
20654310, 20656816, 20662060

Introduction

Sorcery is a computer game written in C++ based on the popular card game *Magic: The Gathering*. The game was written from start to finish using the C++ standard library, and managed with the git source control tool. Sorcery was designed with robustness and resilience to change in mind—the code was written to have low coupling and high cohesion simultaneously. This was done by using the MVC architectural style, with several other common and widely-used design patterns used for specific components of the game. The game can be played both through a command-line interface, as well as an optional graphical user interface. In creating this game, we ran into several challenges. These challenges ranged from being able to work on different things on the same game simultaneously to time management across individual tasks dependent upon completion of previous tasks.

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.

The major difference in the implemented design and the original design was splitting the Player class into a PlayerModel containing the data and PlayerController which owns that data, manipulates it, and communicates with the rest of the cards through BoardModel. This is more consistent with the MVC design, since the data and manipulation of the data is split into separate modules. The only other changes were adding additional public member functions for specific classes where necessary.

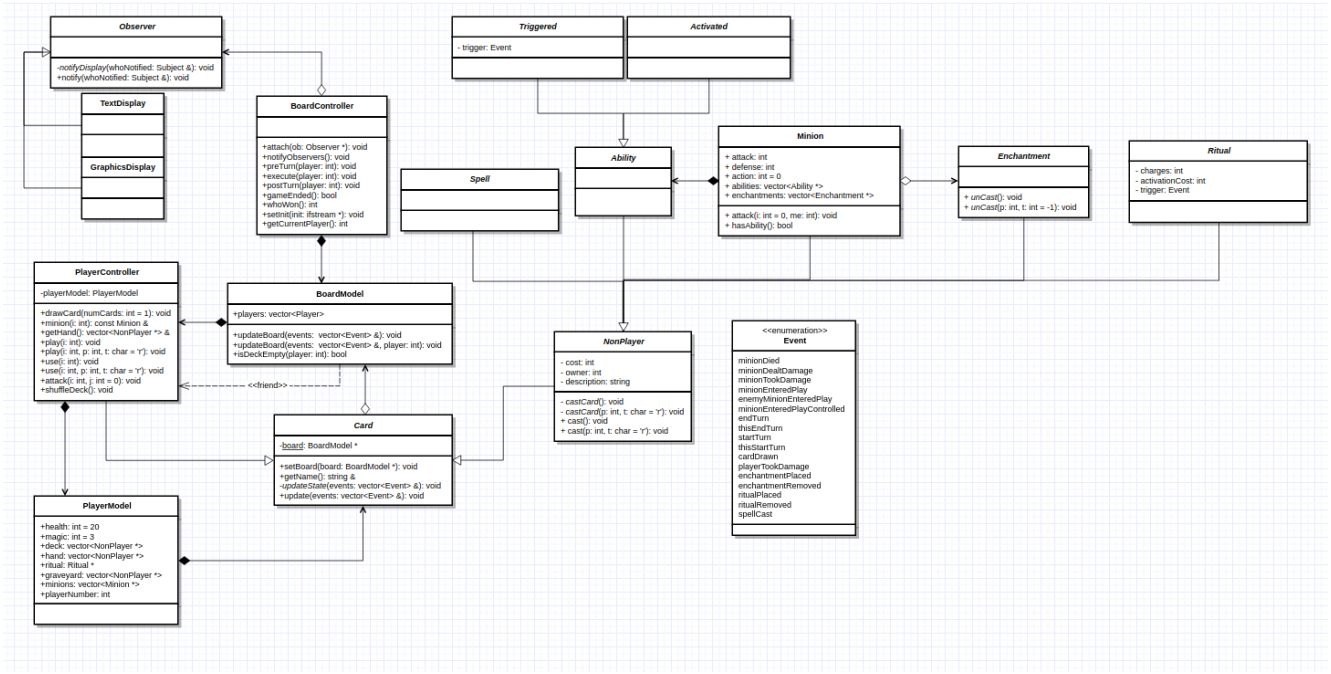
Upon initializing the data, the BoardModel stores the data of the two players on its stack. The players' data consists of the PlayerModel class, which contains fields for the players' health, magic, and pointers to their respective cards; this class is wrapped in PlayerController, which handles manipulation of this data. The physical cards are always allocated on the heap, and PlayerModel uses smart pointers to access these cards. The smart pointers are moved around in various lists (represented by std::vector) to represent movement of cards from the deck, to the hand, the field, and the graveyard.

The cards are first constructed by passing the name of the file containing the deck's data to its respective player. This data is a list of filenames, where each corresponding file contains a card's data. The player then iterates through this file, and constructs each card with the data inside the file. The pointer to each newly constructed card rests in the list representing the player's deck, as all cards must start from the deck. After the entire deck is constructed, it is shuffled randomly.

The cards themselves are organized in a top-down hierarchy starting with an abstract class, *Card*, which allows every card access to other cards' and players' data. Card breaks off into two direct subclasses: PlayerController, and NonPlayer. Every card played in the game therefore inherits from NonPlayer as either Minion, Spell, Enchantment, Ritual, or Ability (which has subclasses Triggered and Activated). Minion is a concrete class, while the other card types are abstract, with each physical card directly inheritting its type (e.g. AddSpell directly inherits Spell). Furthermore, Minion employs the decorator pattern, where the decorators are the triggered and active abilities that Minion is enhanced by.

In order to allow for the easy addition and removal of "views", every display in Sorcery inherits from a base abstract class called Observer. In essence, the subject-observer pattern is employed wherein the subject is the BoardController, which notifies the displays about what information was requested. As a result, each observer, or display, is notified and able to handle its self-contained display logic. In the future, for example, if any new graphical interfaces were added, it would require absolutely no change to BoardController as will be discussed later in this report.

UML



Design

From a glance, the design follows the MVC pattern. The BoardController acts as the main controller that handles transmission of data (the "model") into the display for the user (the "view") using a modified Subject-Observer pattern, where the BoardController acts as a subject (but it does not inherit an abstract subject, because there is no other concrete subject that would make sense to have), and TextDisplay and GraphicsDisplay inherit an abstract observer. The BoardController owns a list of observers, and it notifies each observer in the render function based on data from the BoardModel. Since the model, view, and controller only interact with each other minimally and responsibility is separated independently, this provides a design with low coupling and high cohesion.

The view, as previously mentioned, is governed by the base abstract Observer class. Each display, namely TextDisplay, which outputs the view in text form, and GraphicsDisplay, which outputs the view in graphical Xwindow format, inherits from the Observer class. In doing so, the subject-observer pattern is used and thus there is an extremely low amount of coupling as the Views are completely separate and require no change from the Model or the Controller to function. In keeping with the MVC pattern, the View is the only way to output data to the user. No other class is outputting text or graphics ensuring low coupling but high cohesion as each of the model and controller must work seamlessly with the View. The implications are clear: due to the way the View was implemented, it can be completely removed and reprogrammed and little to no change would be needed to the other aspects of the code.

The model, which is wrapped in BoardModel, consists primarily of the players and their card data. The players are stored on the stack of the BoardModel, as a vector of unique pointers to PlayerController, which owns an instance of PlayerModel on which the data is stored. This is a private field, which is accessible by BoardModel through friendship. 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 PlayerModel's stack. Shared pointers to these 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 Card class 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 (called through the public *update* method) 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 occurred (by reference), so that every card can act accordingly.

PlayerController directly inherits Card. In doing so, PlayerController 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 occurred. 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 the other direct subclass of Card, NonPlayer. NonPlayer provides public getters and setters 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 command pattern, where Minion acts as a receiver, BoardController is the invoker, and each Ability is a concrete command. However, Minion *owns* Ability (instead of BoardController), since a minion's ability moves with the minion wherever the minion moves throughout the game (including to the graveyard). 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.

Resilience to Change

Throughout the entire development of the game, one question always prevailed: "how easily can we accommodate change?" In other words, we created the code such that adding new features in any part of the game would, overall, require the least amount of code and architectural changes possible. Such features can be broken up into the major different aspects of the game that would potentially change: the display, the number of players and the interactions between them, the game rules, and the cards themselves (modification or addition of new cards).

To start, a change in display is easily accommodated because the only class that interacts with it is BoardController (the main function simply instantiates it)—however, BoardController does not actually know what type the display is, and it doesn't need to. All it needs to do is pass in the correct data from the model side of the application to its list of displays (i.e. observers), and the display will handle the rest. Thus, adding new displays (e.g. 3D graphics, split screen, virtual reality, etc.) will be relatively easy to integrate into the game. This is also an example of low coupling, since the view is only dependent on the controller for getting the right data, and it shows high cohesion in that the view and BoardController work towards the same goal: displaying data to the user.

In addition, the BoardModel, which wraps the player and card data in one class, allows for the flexibility of adding additional players with additional cards, since it holds a vector of players. The rest of the rules remain the same, only that the main game loop must now iterate through the additional players. The view is also left unchanged, since the displays don't care which player's turn it is, or which player owns which card—it simply takes in data, and displays it. As well, improving the interaction between players (for instance, by using separate controllers as opposed to the same keyboard) can simply be changed in the main function which takes in raw user input to process, and then send it to the BoardController. Again, this shows that through a highly cohesive design where the individual classes are minimally dependent on each other (low coupling), and the code is resilient to changes even as major as changing the way users interact in addition to the number of users that can play one game.

The game logic is split up between main, BoardController, and the implementations of the cast function. While this may seem coupled, from a closer look, it is evident that main simply iterates through the players to call the pre-turn effects, then the turn itself, and finally the post-turn effects—it therefore handles *only* the logic of the order in which things happen. What actually happens is dealt with in BoardController, which determines what needs to be done. Thus, changing the order of the game or changing what actually happens in each turn means that we only need to change one part of the code. Moreover, if we wish to change the functionality of a specific card, the only change that needs to be done is that card's specific implementation of the cast function; everything else remains the same, especially since each card has full access to other players' and cards' data through the static pointer to BoardModel.

This leads us to the modification and/or addition of cards. Modifying them is a simple matter of changing field values in the concrete implementation or modifying the cast function implementation, which changes the usage/effect of that card. Adding new cards in a deck is a simple matter of creating the concrete classes for each of Spell, Ritual, Enchantment, and Ability, inheriting from its respective

type, and then implementing updateState and cast. For the minion, all that is required is a unique name and its own attack and defense fields. The rest of the code need not change—the logic and execution of the players' turns remains the same, and the display does not change either, since all it needs is the card's data so that it can display it.

Questions

Question 1: *How could you design activated abilities in your code to maximize code reuse?*

Activated abilities in our project were designed as an inherited class from the abstract Ability class; this allowed us to inherit common functionality between abilities and all other cards in general such as casting and updating the card. Since activated abilities are a separate subclass of the Ability class and each minion has a list of abilities, we were able to store both triggered and activated abilities within one list in each minion. As a result, we did not have to write separate handlers for the distinct types of abilities which effectively maximize code reuse. In addition, if new activated abilities need to be created, we simply need to implement a new concrete Activated Ability class. The biggest code change required for creating/modifying an activated ability would therefore be the implementation of the cast function.

Question 2: *What design pattern would be ideal for implementing enchantments? Why?*

In our design, we used a modified command pattern. In our case, each enchantment is a concrete command and each minion has a list of these commands. Thus, when the caller/invoker (which is the BoardController) wants to add an enchantment to the minion (which is the receiver), it is as simple as adding an enchantment to the minion's list. As such, all modifications that an enchantment applies are handled. Furthermore, this design is effective because it decreases coupling. That is, adding a new enchantment means we only need to create a distinct concrete enchantment class. Like our activated abilities, no other changes are required to introduce a new enchantment card.

Question 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?*

Using our initial implementation where the minion has its abilities as a property, to accommodate for this change in space, our design implements a vector of abilities within each minion. Since both triggered and activated abilities inherit from the Ability superclass, this vector can contain any combination and number of both types of abilities making this implementation very dynamic. Our current design pattern maximizes code reuse because if there is a new type of ability apart from triggered and activated abilities, we simply need to create a new class that inherits from Ability and no further changes are needed. Implementing a new concrete ability of any type only requires the creation of a simple function.

Question 4: *How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?*

We used the subject-observer pattern and added each type of display required as an observer and the BoardController as the subject. Thus, at every stage of a turn: pre-stage, execute-stage and, post-stage, each observer is notified to display the new state of the board. In this way if a new interface is created we can simply add it to the list of observers, this way no other changes are needed.

Final Questions

Question 1: *What lessons did this project teach you about developing software in teams?*

The first thing we noticed in working on a team is that development progress is significantly faster. In our case, since we had three different programmers with essentially equal competence, we had triple the programming capabilities of just one programmer. Thus, the individual tasks that needed to be done were finished, on average, three times faster. In our case, working in a team was especially efficient because we used the git source control tool so that we could all work on the game simultaneously, while on different machines.

However, we realized that communication was absolutely critical. Every single one of us needed to understand exactly what the other teammates were working on/fixing so that we would not have two developers trying to fix/implement the same thing in different ways. This was not a problem for the most part, since the tasks were split up so that we worked with as little overlap as possible. Then, because the code had low coupling, we had very few merge conflicts and whatever ones appeared were easily dealt with.

Finally, we also realized that to harness a team's full potential in the most time efficient way possible, we had to utilise our individual strengths. Jafer has extensive experience using git, and this helped us clear any source control issues quickly; Berges has extensive knowledge of *Magic: the Gathering*, and so he understood the rules and game logic like the back of his hand; Catalin has previous experience working with the MVC pattern, and thus he ensured that both the design and implementation were robust and resilient to change at all times. Ultimately, our individual strengths and passions were used as powerful tools to develop a high-quality, complete application.

Question 2: *What would you have done differently if you had the chance to start over?*

For the most part, the project went through very smoothly. However, there were a few things we wish we would have done differently. First, Berges regrets not having used a Linux machine: the rest of us were using Linux and we frequently ran into Linux-Windows compatibility issues that would not have existed had Berges used Linux to begin with. Second, our time management worked quite well, but we still ended up leaving some crucial parts to the end, which put a lot of pressure to finish development near the deadline, and that often had us trying to find shortcuts instead of adhering to good, efficient design. As a result, we would have wanted to start working on the key features of the game much earlier on. That way, we would also have had more time to implement extra features as well.

Conclusion

Under a two-week timeframe, we were able to design and implement *Sorcery*, fully functional and complete. The design ultimately remained very similar to our original design, and all of the challenges we ran into were dealt with successfully. We developed many skills with regards to working in a group—both technical, and soft. In conclusion, *Sorcery* is overwhelmingly successful, and on its way to winning the 2017 Game of the Year Award.