Daniel Zhang, Maggie Liu, Stanley Lin

dc3zhang, mm2liu, s38lin

CS246 A5 Final Design

Introduction

The following design document outlines a complete overview of the Sorcery final project completed for CS246 in Fall 2023. In this outline, a comprehensive report will be given regarding the overall structure and design of the program, as well as the challenges, successes, and extra implementations that were included along the way.

Overview

The main program is run from Gameboard.cc. As players of the game, you can pass in arguments including *-init filename* where the program will take the commands and initialise the game, *-deck1 filename* and *-deck2 filename* which will supply the cards for the decks of player 1 and 2 respectively, -testing which will set the isTesting local variable to true, and the -graphics option that creates an external graphics window which updates as the game progresses. Thereafter, players are prompted to enter their names if the *-init filename* was not provided, and the game initialises by loading the decks of each player from the database, based on the file of card names for decks. We used a while-loop for the game loop and we continuously ask for a command from the user. We used XWindow for our graphics and implemented elements of the observer pattern for our GraphicsObserver class to constantly update the graphics. Our text interface uses the provided ascii_graphics class and methods we wrote to create vectors of vectors of strings to print the board and hand, and to inspect a minion.

The Player class contains a player's deck and hand, which are vectors of shared_ptr of Card objects, and a player's minions and graveyard, which are vectors of shared_ptr of Minion objects. It also contains a shared_ptr to a Ritual object, which the nullptr if no ritual is active. Each player has lives and magic, and the Player class controls the cards played and actions made by the player. This is also where we read in cards from card.data and shuffle the deck for a player: we created a map<string, shared_ptr<Card>> that maps names to a card to allow us to read in any deck from a text file, allowing for flexibility. To allow for duplicate cards, we wrote

copy constructors for our card subclasses that take in a shared_ptr whenever we were loading our deck using the map to cards.

The core of this project revolves around the card classes. We have an abstract card class that the Spell, Enchantment, Minion, and Ritual classes inherit from. It serves as a base class as it encapsulates common attributes and behaviours for the different types of cards including name, type, cost, description, and owner (which player it belongs to). It employs polymorphism through a virtual play method, allowing subclasses to provide specific implementations for playing the specific card and card-specific error handling. It uses encapsulation via getters/setters and private fields for data integrity and abstraction.

The Spell class inherits from the card class and the play method is overridden and overloaded via play method that takes two integers–a player and a target (active minions or ritual). The Ability class for activated and triggered abilities “has-a” Spell, which will be discussed later on, and allows for the reuse of code. The Ritual class has a has-a relationship with the Ability class since it behaves similarly to a triggered ability. Thus, the Spell class does a wide range of things in the play method depending on the ability or spell or ritual. It has access to both player1 and player2 and depending on the name of the spell card, it is able to summon minions, recharge a ritual, damage minions, enhance minions. etc. Thus, when a user types in the use or play command to play a spell, minion with a triggered ability, or ritual or to use an activated ability in Gameboard.cc, these all end up calling the Spell class’ play() methods.

Enchantments inherit from the Card class as well. Minions have a vector of shared_ptr to an enchantment which directly mutates the fields of the respective minion. This includes modifying the minions ability cost, increasing the attack/defence, increasing actions per turn, etc. This mutation occurs whenever an enchantment is played and attached onto a minion. The opposite occurs when an enchantment is removed.

Design

One of our main challenges when implementing the planned design was the use of design patterns, especially when it came to the enchantments. At first, the plan was to implement a decorator design pattern for the enchantments class, as it would be dynamic and applicable to individual objects, which in this case would be the minion that the enchantment is applied upon. However, after careful consideration and a plethora of testing, it was determined that a decorator pattern, or design pattern in general, would be quite inefficient and unnecessary in this circumstance. This issue stemmed from the fact that Enchantments inherit from the Card class, but also must be applied upon a Minion to alter its fields. Thus, when trying to implement the decorator pattern, we encountered challenges in preserving the integrity of the inheritance structure while dynamically attaching behaviours to the Minion instances. The intersection of Enchantments and Minions posed a unique set of complications that was better addressed by simply giving each minion an enchantment field held in a vector (since enchantments are stackable), and making Enchantments a standalone class that inherits from Card, such that when an enchantment is played upon a minion, the changes in att/def/abilities are directly altered in the minion class depending on the enchantment applied, rather than making the enchantment a decorator of the minion.

In addition, we also developed the software to have high cohesion and low coupling. For instance, we designed Ritual with a ‘has-a’ relationship with Ability, since each of them have a triggered ability, which can be activated by triggers and the charges that are unique to Ritual. Ability also ‘has-a’ spell, since they all change the game in some way. Thus, implementing Spell results in high cohesion in the code. There is also low coupling in our code since we create relevant public functions in classes, such as attack and decreaseLife in the Player class. As such in GameBoard, when it calls on Player::attack(), it does not depend on the implementation of attack due to abstraction. Within the GraphicsObserver class, the functions for drawing descriptions, drawing cards, and removing cards are also abstracted from the larger function that draws the entire board. This allows other parts of the game to call these functions to update a specific part of the gameboard on the graphics display.

Resilience to Change

The design supports the possibility of various changes to the program specification due to the high cohesion and low coupling design. For example, if the program specification changed to minions not going to the graveyard when they are depleted of their defence, this can be modified within the Minion and Player class and would not require other classes to be updated much. Our design also supports the addition of completely new cards into the database and deck files. We are reading in card information from a text file, so merely adding a new line to the text file would be enough to introduce a new minion to our game. If the minion had an ability, we would be able to just make a change to the spell class since we would be able to read in the trigger type of the ability (1 being start of turn trigger, 2 being minion enters trigger, 3 being minion leaves trigger, 4 being end of turn trigger, 5 being an activated ability, 0 being no ability) and then the ability would play its associated spell card. Since the implementations of spells, abilities, and rituals are all within the Spell class, it would be efficient to add a conditional check for playing the new card. If there are multiple similar spells, new helper functions can be created in the implementation file to reduce code duplications. Thus it would be easy to add new minions, spells, and rituals to our game.

Moreover, our design moves away from hard-coding enchantment multipliers as much as possible. We allow values other than the standard +2, *2 multipliers to be read in for Giant Strength and Enrage, so if the card info changed the values of the multipliers to +4 or *6 for example, the enchanted attack and defence would accurately add 4 or multiply 6 when applied on a minion. New enchantments would be easy to add as it would only involve a change to the Minion class and a new line in our card.data text file. New triggers for triggered abilities and rituals would be easy to implement as we used the observer pattern. We would only modify appropriate code in the Player class to notify the activateTrigger (notify) method in the GameBoard class which would use the appropriate ability.

New input syntax would only involve changes to the Gameboard main game loop function but all the other classes for the various cards would remain the same. Furthermore, since each class is self-contained and encapsulated, changes to the behaviours of each card only involves the card

type's class to be modified. Thus, our design exhibits high cohesion within classes and low coupling between them.

Answers to Questions

Q: How could you design activated abilities in your code to maximise code reuse?

The Ability class re-uses the Spell class since they work similarly: the only difference is that spells have a cost but abilities are merely spells with a 0 magic cost. It has a has-a relationship with the Spell class and it can use its shared pointer to a spell's play method for both non-targeted and targeted abilities. More specifically, the Apprentice and Master Summoner's activated abilities re-used the summonMinion(shared_ptr<Minion> minion) method to summon air elementals on the board which is called by the Spell class. Finally, the Novice Pyromancer was able to reuse the decreaseLife(int amount) method in Minion and the play(int p, int t) method in Spell for its activated ability.

Q: What design pattern would be ideal for implementing enchantments? Why?

We decided against the decorator pattern for implementing enchantments, as detailed in the Design section of this document. The decorator pattern is rigid: while it is able to attach decorators to a component dynamically, it is difficult/inefficient to remove them. In our project, when minions with enchants are unsummoned or a disenchant spell is played, we needed to be able to remove enchantments which was not possible with the decorator pattern. As well, since every decorator is-a Component and every decorator has-a Component, we realised that an enchantment being a Minion doesn't make any sense. Ultimately, we opted for a vector of shared pointers to an Enchantment class that modified the fields of the Minion class. This made it easy to dynamically add and remove enchantments.

Q: 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 maximising code reuse?

A: The observer pattern would be the best choice for abilities. We used elements of the observer pattern for triggered abilities to best fit the sorcery project. As mentioned earlier, activated abilities reuse the Spell class' play methods, but so do the triggered abilities. Essentially, the

gameboard is an observer of each player's minions, the subjects. Whenever a minion is summoned (enters) and when a minion dies, the gameboard's activateTrigger(int trigger) method is notified and the appropriate triggered abilities are activated. Furthermore, the gameboard also observes itself and whenever a player's turn starts or ends, the same method is notified and the appropriate triggers also activate. This allows for flexibility in the number of triggered abilities that a minion has. To allow for any number and combination of abilities, we could store the abilities as shared_ptr in a vector and whenever an ability is activated in activateTrigger(int trigger), we could loop through the vector and notify the appropriate triggered ability. Activated ability would work as they normally do as we would just require the user to specify which ability they want to use in the use command.

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

A: Observer pattern can be used to implement the interfaces. These interfaces would be observers that are notified when the game board, which is the subject, changes so that the most recent updates to the game are appropriately displayed to the players. Whenever a minion is summoned, a card is drawn/discarded/played, an ability is used, a minion attacks, a minion leaves play, or a turn ends, the interfaces would be notified and would update accordingly. This requires minimal changes to the rest of the code because only notify() calls would need to be added to the appropriate places in the code.

Extra Credit Features

We only used smart pointers and STL containers for the memory management in our sorcery project. We did not have any memory leaks and we did not make any delete calls. More importantly, there are very few raw pointers in our program: only in Gameboard.cc did we have pointers to player1 and player2 but this is not to express ownership, only to differentiate between the active player and the inactive player, as well as for the GraphicsObserver. For the entire program, we used unique_ptr and shared_ptr for our graphics observer and for all the cards.

This was challenging as we had to change the way we were casting because with smart pointers, we had to use std::dynamic_pointer_cast with smart pointers which is a much more modern and

safer approach to casting. Using type casting was crucial for our project as we had an abstract Card class from which the different card types inherited from. It was important for us to ensure that we checked if the cast was valid (and didn't return a nullptr) and that when we used shared_ptr, there weren't any circular references that would result in memory leaks.

Final Questions

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?

We learned how essential it is to have good communication skills when developing software in teams. We were able to be efficient and complete the project on time by splitting up tasks within the team, and setting goals for the tasks, while updating each other on the status of the project. By asking each other questions, we were also able to have more perspectives for potential solutions and were overall faster to achieve our goals, compared to if we were to have completed the program ourselves. This project also taught us useful strategies of separating code changes into smaller pushes, such that other team members can retrieve latest changes faster and prevent complex merge conflicts.

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

If we had the opportunity to start over, we would approach design patterns differently. Instead of treating them as rigid requirements that our code had to fulfil, we should've used aspects that made sense for our project needs. In particular, we should've taken a more intuitive approach to decorators and considered if the cons outweigh the pros when implementing enchantments. As well, we would've considered early on that we could just take aspects of the observer pattern to ensure the efficiency of triggering abilities and rituals, and to implement the graphics observer. We initially made things more complex than they had to be because of this and we could've avoided quite a bit of hardship with a more flexible approach to design patterns.

Conclusion

In conclusion, we were able to implement all the aspects of the original game while including the bonus memory handling challenges with smart pointers and STL containers. If we were to expand on this game, we would implement more cards with unique abilities and improve upon

the graphics. Ultimately, we are extremely satisfied with our end product and found many opportunities for growth through designing, implementing and testing our program.