Final Design Document - Sorcery

Sarvesh Gambhir and Tapish Jain

Introduction

This document will outline and discuss our object oriented project for CS 246 Sorcery. More specifically, we will first give an overview of our program and discuss the various techniques we used to solve certain design challenges. Next, we will describe the flexibility of our design and how it supports changes to program specification. The final talking point will be the bonus features we implemented, as well as a final reflection on the project.

Overview

Our project is structured around the gameboard class as this class is what essentially controls the flow of the game, since all commands inputted by the user go through this class. Essentially, main calls functions in the gameboard class, which then using various programming principles executes those commands. Simpler functions such as attack and play, are all handled in the gameboard, by using the members of the player and respective card objects. However, for more complex functions that require the use of activated and triggered abilities of cards, an observer design pattern is used along with a set of notification scopes for each card, where the gameboard notifies the cards that correspond to the specific notification scope being called (i.e. each scope corresponds to an event in the game, such that when the event occurs, the gameboard notifies the appropriate cards). This will be talked about more in the Design section. This project is also structured around the use of inheritance, specifically when creating the various cards. The basic structure is that we have a parent card class, followed by children classes Minion, Spell, Ritual and Enchantment, which are further followed by grandchildren classes corresponding to the specific cards.

Design

Inheritance:
We used this OOP concept to employ code reuse. This allowed us to substantially reduce the code written for the various card classes, as we recognized that all cards shared various attributes such as name, type, description, cost, owner and notification scope. Note that the notification scope was used for the observer and the owner (an integer storing the number of the player that the card belongs to) was used for cards that dealt damage to their opponent. Next, we realized we could further create sub classes based on the type of card, as the specific card's pertaining to a type all shared the same properties with different implementations.

RAII Principles and Smart pointers:
Smart pointers significantly improved program efficiency and lessened code reuse. We mainly used unique pointers in order to avoid the possibility of cyclic reference problems with shared pointers. The areas where these were used include the storing of two unique player pointers in the gameboard class, storing unique card pointers in vectors in the player class (i.e.

corresponding to the deck, hand, grave, and minions in play), as well as a unique card pointer storing the ritual card in play for each player. We also had a vector storing unique card pointers in the parent card class (in order to implement the decorator design pattern for enchantments on minions). The use of unique pointers meant that we could heap allocate a lot of our objects without needing to manually delete them in the destructor for each class. Since we needed to use multiple card vectors, this would have meant that we needed to repeat the process of deleting the heap allocated objects in vectors multiple times over. Furthermore, using RAII principles, we tied the lifetime of objects to their appropriate owner. For example, all the cards are owned by a single player (as mentioned before, each card stores an integer referencing the number of the player that owns it), and both players' lifetimes are tied to that of the gameboard. Therefore, using smart pointers, as well as RAII principles, we significantly simplified our memory allocation and deletion such that objects were automatically heap allocated at the beginning of the program, and were automatically deleted at program termination, with each object's lifetime tied to another object that owns it.

Public members:
Based on how certain card containers behaved in the game, we decided it would be appropriate to make most of our card vectors in each class public. Our justification is based around the idea that any container that can be modified during the game by the user, could also be public within the design as it improved efficiency when implementing certain card effects. For example, in the player class, we decided to make the card vectors representing the hand, grave, and minions (in play) public, as the game itself had cards that allowed the user to shift cards between these three places (i.e. "Unsummon" moves a minion from the board to the player's hand, and "Raise Dead" moves a card from the graveyard to the board). Furthermore, commands such as "play", and "discard" also allowed the player to shift their hand and board. This also applied to the modifiers vector in the card class, as well as the unique pointer storing the ritual in play for each player. In both these cases, the containers (or the card in the case of the ritual) can be affected by the user and their actions in the game, and so we made them public within the program. On the other hand, the vector storing a player's deck was made private as this container cannot be affected by the users within the game. The deck is essentially created at the beginning of the program, and drawn from at the start of the game, and at the start of every turn. And while the "draw" command can also be used to move a card from the deck to the hand, it is only available in testing mode, and therefore does not justify making the vector public.

Abstract classes:
This concept was used to better achieve inheritance and organize child classes. By not implementing functions like use_ability, cast_spell and notify, all the child classes (Spell, Minion, Enchantment, Ritual) could inherit from the parent Card class and then implement the functions it needed in their respective .cc files. The same applied for the specific cards that inherited from the card types.

Observer:
The observer design pattern allowed us to efficiently implement triggered abilities. We did this by using certain notification scopes that were a member of the card class. Then we would call

the notify_observers function in gameboard.cc and pass in the notification scope and player needing to be notified as arguments to notify these cards to use their triggered abilities. We would call the notify_observers function when the specific notification scope presented itself (i.e when a minion entered play). The notification scopes were strings in the card class and the scopes we used were death, minionEnterOpp, minionEnterOwn, minionEnter, endTurn, startTurn and none. An example of using the notification scope is when an opponent's minion entered play, we'd call notify_observers with the minionEnterOpp scope passed in as a string parameter and the opposing player passed in as an integer i(if the active player was player 1, we'd pass in 2 and otherwise pass in 1). We'd then use a getter to get the player object that we needed to notify and loop through the appropriate player's minion vector and notify minions if they have the minionEnterOpp scope. In this case we'd notify all Fire Elementals and in fireElemental.cc we included the appropriate implementation for the notify function.

Decorator design pattern:
This technique was used to add enchantments to a specific minion. Essentially a vector of enchantments called modifiers was a member of each minion class. The modifiers vector was used to store the enchantment cards from oldest to newest (index 0 representing the oldest enchantment and the last index representing the newest). Each enchantment essentially had a modify type that specified what change it made to the minion it was attached to. For example, for the enchantments that modified the minion's attack and defense values, we specified them with stat_mod/atk_add and stat_mod/atk_mult. Furthermore, each enchantment had accessors that returned the amount that their effect modified the minion by. So when the gameboard, or the display called getAttack() on a minion, the minion's accessor would iterate through its modifiers vector, and based on the enchantment's modify type, it would call the appropriate getter from the enchantment (i.e. getAttack()), and either add or multiply its current attack value by the value returned. Some enchantments' effects could be implemented with just the modify type. For example, "Delay" was specified with the "no_ability" type, such that at the beginning of each turn, the function minionsReset() in the player class would iterate through each minion's modifiers vector and check if any enchantment had that type. If so, it would refuse to give it an action point.

<u>Resilience to Change</u>

First, in our design we made it a priority to compartmentalize code into functions through the use of helper functions to enhance code reuse, so if changes to the program specification needed to be made they could be done easily. For example; in the Display class when outputting a board several helpers were used so that if the board's display was to change this could easily be accommodated in the implementation. A specific example of this is if per row we wanted to display more than 5 cards. This could easily be done by simply sending in a different upper and lower bound (representing the number of cards per row) to the outputRow function, meaning that the programmer wouldn't have to go in and significantly change the implementation of any functions.

Next, we designed our program in a way that allowed the easy addition of new cards. New cards could be added by simply creating a new class for that specific card and not having to change the implementation in other classes. We achieved this first through inheritance, meaning the new card would already inherit the properties of a basic card and more specifically the properties of the type of card. In addition, by using notification scopes and limiting the use of card names to achieve functionality, any game effect that the card would initiate would only need to be defined within its own specific class, and no other area within the program would have to be altered. This was done by giving each card a specific notification scope and then using the observer to notify the card in regards to when it needed to use an ability. Further, if more enchantment cards needed to be added, due to our use of a decorator and modify types as specified in the design section of this report, nothing in the implementation would need to be changed regarding the functionality of attaching the cards to a specific minion.

Furthermore, if in the future more triggered abilities needed to be added, this can easily be done by slightly altering the observer. In fact, the current notification scopes cover most possibilities regarding when abilities are triggered, thus adding additional triggered abilities would mean that you would just have to specify the appropriate notification scope for cards with said abilities. If a new notification scope was needed, this could be accommodated by simply notifying cards with such a scope when needed, and then in the specific card class itself implementing a notify method that's already an abstract function for a card. Thus, no changes to the implementation of the observer's functions, functions in the gamboard, player, display or card class would be required.

Finally, we compartmentalized all of our text output into a single class called "Display". This allows for possible changes to the game display to be done efficiently without having to change the fundamental design or structure of the program. Any new display type can either work off of the current display class, or can be defined separately and then called when appropriate from gameboard. This would be as simple as adding a few functions in gameboard that called the appropriate methods in the new display to achieve the correct output.

Answers to Questions

Question: How did you design activated abilities to maximize code reuse?

Each minion card defined its own use_ability function that was called by gameboard after the appropriate error checks. For example, if the minion had no activated ability, the use_ability function did nothing. On the other hand, for a minion such as Master Summoner, the use_ability function carried out the appropriate functionality. This maximizes code reuse as any additional minions that we might want to add do not need to change the fundamental design of the program, or any other functions in gameboard. It would essentially just define its own use_ability that would be called by gameboard when appropriate.

Question: What design pattern did you use for implementing enchantments? Why?

We used a decorator design pattern because this allowed us to attack multiple enchantments to a single minion and apply the enchantments effects to a single minion based on their position in the vector of enchantments used by the decorator. In our decorator the minion class was the component, the specific minion card was the concrete component and the enchantment cards were the concrete decorators. Using the decorator design pattern allowed for easy addition and removal of enchantments from minions. Essentially when a function such as getAttack() or getDefense() was called, it would loop through the minion's modifiers vector (essentially the enchantments vector), and appropriately alter the value returned. Other enchantments had an effect outside the scope of the minion class. For example, in the player class, when the player's minions had their actions reset, the minionsReset() function would loop through each minion's modifiers vector to check if a "Delay" card was present. If so, it would not reset its action points.

Question: 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?

A decorator design pattern would help us achieve this while maximizing code reuse. Similar to enchantments, the minion would act as the concrete component, and we could create concrete decorator classes for each triggered or active ability we wished to add.

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

The observer design pattern could make supporting multiple interfaces at once easy requiring minimal changes to the rest of the code. Essentially each interface will be implemented in a separate observer class that would be notified by the game board when called. If a player wished to use a specific interface, the display functions in the game board would essentially notify the appropriate observer and the observer would take care of outputting the interface to the user.

<u>Changes to UML</u>

The updated UML diagram is essentially a more refined version of our original. As we implemented the program, we realized we needed or did not need certain functions or fields. For example, instead of storing player vectors in card for the owner and opponent, we felt that it was more efficient to store the player numbers instead, and add an accessor in gameboard so that the card class could send in a player number to the accessor and retrieve the player pointers. Another change we made is to the display class, as instead of having a display vector for both players, we instead opted to go with one board vector because this would allow us to more easily store the cards in play for both users. In addition, we added an inspect minion function in display to print out a specific minion and its enchantments. Originally we were going to do this in the minion class itself, but realized it would be better design to output everything from one class in case we wanted to expand our program to include graphics or another interface. This is because we’d then only have to tweak the implementation of the display class, whereas in our original design we’d have to change the implementations for both the display and minion class. Also, we changed some of the functions in display to allow for better code reuse. For example; by adding the outputRow function if we wanted to add more rows to the board or increase the number of cards per row, we’d simply have to change the arguments passed to the outputRow function from outputBoard (refer to display.cc). Lastly, since in all our vectors (hand, deck, minions) we were storing the cards as a card unique pointer and not a specific type of card, we were forced to move the cast_spell and use_ability functions up to the card class and make them virtual, instead of just declaring said functions in the classes they were needed. Despite all this, the high level relationships between classes remain the same, as well as the overall design of our implementation.

Extra Features

Handled all memory using STL containers and smart pointers: We employed the use of unique pointers to store our player and card objects and used vectors to store the deck, hand, minions in play, display and enchantments on a specific minion. The biggest challenge we faced with this was when trying to update the members of a player, which included updating the hand and minion vectors. We solved this by using a getter in the gameboard class which would return a pointer to the player using the .get() method. This would then allow us to modify the player objects using the classes public interface in classes other than gameboard and player.

Error handling and communicating with the players: We made sure to include error handling for all scenarios that could lead to undefined behaviour and communicate the reason for throwing said errors to the players. This is important as the player will know exactly what the consequences of their moves are and what the state of the game is at any specific time. In addition, it will allow the players to better understand the game and not have to make assumptions about behaviour not specified. The most challenging parts of this were organizing our code to not repeat errors, ensuring we weren’t throwing errors in unnecessary places and making sure we were throwing errors in all places the program was undefined. We solved all the aforementioned challenges through extensive planning and testing.

Game Decisions: Certain decisions were made regarding behaviour not specified in the project specifications. For example; if the user passes a deck file that’s invlaid or does not open, the program will use the default deck for the player and display a message outputting this information. Also, for the play and use command if the player passed in targets when they weren’t needed the program would run the command as if no target’s were passed in and output an appropriate message to the user informing them that no targets were needed for the command they entered. Another design decision made was in the unsummon card (in unsummon.cc) where if the user’s hand was full the target minion was instead sent to the graveyard. Also, if a player's deck, hand and minions in play are empty then the other player will be declared the winner. The last design decision made was in unsummon (unsummon.cc) when this card is played against an opponent and their hand is full. The minion card would then be sent to the opponent's graveyard with all enchantments and attack/defense intact.

Final Questions

1) We learnt that planning is the most important step. This includes planning your overall design and planning how you're going to split up work, so that all members of the team can work on parts that don't overlap with each other. This results in less debugging when components group members are working on are combined, meaning less time is wasted making changes that could have easily been avoided. Since overlap will happen regardless it is important to push, pull and compile code frequently to tackle bugs before your code gets too large, making it more difficult to find the root cause of errors. This means that testing, specifically doing unit testing when components different group members have done are combined is essential. This again allows you to catch bugs before your code base becomes very large and convoluted. Lastly, we learnt that communication is very important, as if one person deviates from the proposed plan it is important to inform the other group members, as they too may need to change their code based on the changes made. In addition, it is important to talk about bugs in the program with other group members as the root cause may be due to code in someone else's work, which means debugging would take less time with effective communication.

2) Even though through the process of completing this project our original plan did not change much because we did spend a decent amount of time planning, if we had spent more time planning, specifically talking about the smaller implementation details we could have decreased time spent debugging and changing our design. In addition, once completed we did not create an in depth test suite like we would have for the assignments. Thus, if we had the chance to start over we should have created a test suite that tests several edge cases to ensure our program is running the way we wanted it to.

Conclusion

That concludes the discussion of our CS 246 Sorcery project and its design and documentation.