Sorcery Game Design

group member: Jia Hao, Chengcheng Hu(c45hu)

completion date: Dec 4th, 2017

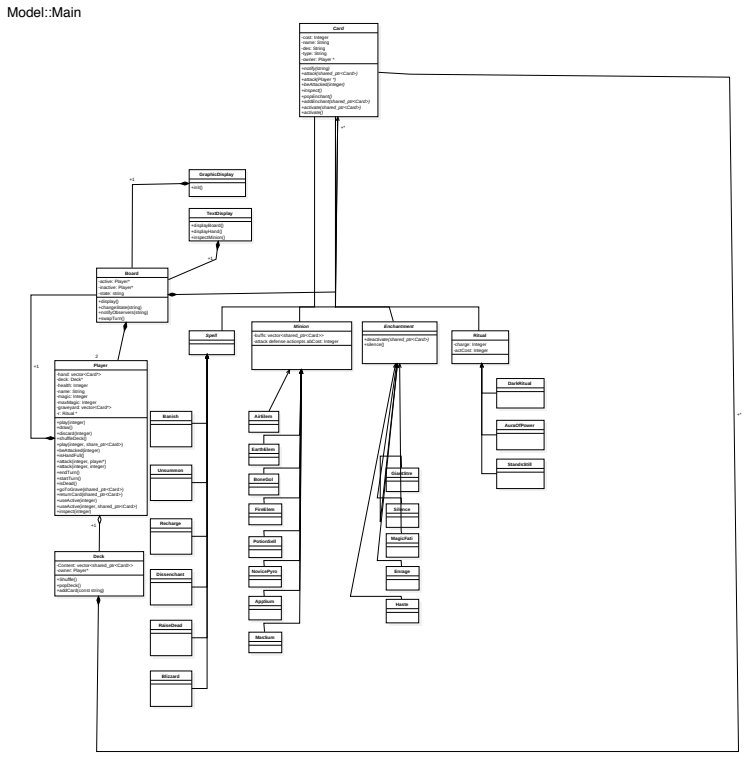
Introduction

It is a Card Game written in C++. Every player holds a number of cards with different functionalities. The game continues as each player play different cards to cause damage to the opposite. The game's objective is to reduce the opposing player's life to 0, at which point the game ends.

Overview

We have basic classes Card, Player, Board, Deck and Display. Player and Deck own Card and Board has Player. These plus main achieve the abstract interaction functions needed in the game.
And we have 4 abstract classes inherited from Card which are Minion, Spell, Ritual, Enchantment, so these all have inherited pure virtual methods. Keep some pure virtual methods and implement them in the concrete subclass inherited from each 4 types abstract class.

Updated UML



Design

Headers

Player.h, Card.h, Deck.h Board.h Minion.h, Enchantment.h, Ritual.h, Spell.h, Sorcery.h

Cards

The card class is defined to be an abstract class so that we can implement pure virtual method to let minions, enchantment, rituals and spell to do different things. Since we have declared pure virtual methods in the superclass Card, we can do the polymorphism for our hand, graveyard and deck which all contains a vector of shared_ptr<Card>.

Each card has a name, type, description, cost and owner. These fields are set by the constructor of Card class. Each card also contains attack and beAttacked methods (mainly used for minion), activate method (used for activate ability and spell) and notify (designed for triggered abilities). Although some subclass may not need those methods (e.g enchantments do not need to be activated), the function was necessary for polymorphic design. We can just implement some empty functions when there is no need for a method.

When the game is initialized, either –deck* filename.txt command or default deck will constructed different types of cards. Since we all use the shared pointer to store cards whether in deck, hand, minion, graveyard, as long as these vectors go out of scope, the memory will be automatically deleted.

Player

Player will be our main class used to do the interactions between user-command and specific card functions.

Players have the following fields: a name, health, magic and maxMagic, deck, hand, graveyard, minions, ritual. Name is only for display purposes. The last 5 fields are shared pointers (or vectors of shared pointers) to existing cards that were created at initialization. Getter and setter functions were added to the class for each field to accommodate encapsulation.

The methods of Players are mainly divided in to parts in order to achieve purposes.

1.Attack and beAttacked

The Attack is mainly used to deal with the attack command, calling the attack method of the required card on the minion of Player. The beAttacked deals with the “attack face” condition.

2. Play (with different parameters)

It is used to play minion from a player's hand to player's minion or play a spell when there is a play command.

3.useActive

This is for use command. With this method as an intermediaries, we can call the activate() (explained later) of the used Card.

4. goToGraveyard and returnCard

This is for the movement of a card between a player's hand, graveyard and minion. This will be used when encountering some special abilities.

Activate & Triggers Abilities

We do not implement an additional class for Activate and Triggered Abilities. In order to see if a minion has an activate or triggered ability we first look at the abCost field of the minion: If it is not 0, then it should have an activate ability; else if the abCost is 0 however the des field of the minion is not "", it means the existence of a triggered ability; the rest condition will simply indicate that the minion does not have neither activate nor triggered abilities.

We implement different activate() methods for each specific minions so when the command is use xx, we will first call Player::use() then call the activate() for the used minion card.

For triggered abilities, we have a state field in the Board along with a changeState() method. As long as triggered events happened, we will call Board:: changeState(string event) to do 2 things:

1. Change the state field of the board;

2. Call the Board::notifyObservers(event) where observers are all minions and rituals which are on the board so that each observers (specific minion and ritual) will call different notify(event) to achieve different functionalities.

Minions

Each minion has additional attack, defence and abCost fields and a vector of buffs. Minions have the option to attack and use their abilities both implemented in the class. So minions have attack functions, to deal simple attack conditions; activate() to use activate abilities and notify() dealing with triggered abilities.

Enchantments

Enchantments are emplaced in a private field of minion as a vector of shared pointers to card. And then give a method deactivate() to deal with the popEnchant functionality.

Rituals & Spells

Rituals have additional fields for the number of remaining charges and an action cost. Ritual mainly deal with the notify(event) function, first see if the event matches trigger requirement if does match, then do the required actions for each concrete rituals.

Spells can only be played directly from the hand and simply call the activate method on the card.

Main

The main function first checks for the command line arguments: player 1 deck, player 2 deck, and initialization file. The decks are then constructed for both players. The program then reads input line by line from initialization file until EOF, then reads commands from cin. For each command, a series of functions will be called based on the arguments. Mostly when the command is attack or use or something related to card, we call the corresponding player methods since player has most pointers of cards which is convenient for us to do operations. Pointers to the active player and non-active player are used to determine which player functions are called when commands are called (play/use). When command is end, we will call endTurn and startTurn for each Players and then swap the pointers of active and inactive players. Whenever a player reaches 0 life, the winner is displayed and the game exits (loop break).

Resilience to Change

Since we design all base classes as abstract classes and we've done all the interaction parts, if we have a new card, we just need to write its own .cc and .h file and implement its override methods.

Also we actually place all the pointer of cards appeared in the game in the player class and board class has pointers of players. As long as we have a pointer of one player, we will get access to all other cards, board, hand, graveyard etc. It would be easier to accommodate new features like recalling a ritual etc.

Answers to Questions

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

We just use the activate() methods to do the activated abilities. Since this is a methods we inherited from very superclass Card all the steps we need to do to connect activate and "use" command are settled down. We just need to implement a new activate() method when there is a use of activated abilities of a new concrete minion.

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

We implement enchantments as a vector of shared pointers in a private field of Minion. Because with the methods in STL vector, we can easily represent the add and pop in the required order.

4.4.2 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. How might you design your code and the user interface to accommodate this?

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

We have all display operations exist in a separate textDisplay class now. So we can have all interfaces exist in separate classes. And then construct an instance in main at the first place. When there is a need to show the interface, we only need to call methods of different class at the appropriate place.

Final Questions

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

Software developing is never an easy thing. It is a huge task that always need corporations. But we have to assign tasks appropriately and figure out a way that the works from different members will not cause conflict.

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

1. Although without specific implementation and features, it is hard to think about the whole running process of the project. It is still worth the pain. Because when we start to write header files, we got a mess using the old UML. We spent a lot of effort to think it over which we could have saved if we done all the things right in the first week.
2. Do tests while writing. If we had the chance to start over, we will definitely follow the advice and test immediately after one module is done. Since we did not keep up this time, in the final compile and running stage, we use a lot of time to figure out which part is wrong.

Difference in UML and Design

1. We delete the subject and observer class in UML2. Before actually implement the project, we think the triggered method and display should work like a4q4. Whenever there is a change in the Board, we notify all the minions. And since some minions need

to interact with other minions, so we thought minions are similar as cells. After we start to implement, by redesigning the private field of players and board we get an easier way to do so since a board contains all pointers of card (by getters). In our new design, there is no strict observer pattern but we design notifyObservers and notify methods which mimic the functionality of observer patterns in board and cards.

And for display, since we do not need to display automatically all the time. So we decide to let main has a textdisplay instance and then call its member functions when necessary.

2.In UML1, we try to implement enchantment as decorator pattern. In new version, we design enchantments as a vector and as a private field in minions. We encounter difficulties when popEnchants using decorator pattern and since we use the shared pointer it would be a little bit safer about memory using the vector methods.

3. In old design, we regard Card class as a really abstract class with no methods in it. In the new version, we realized that we need to do the polymorphism, we have to move all methods in the subclass in old design to the card class as pure virtual methods.

Schedule

Phase 1 (Nov 27-29):

Jia Hao: Create a main harness. Create minion class.

Chengcheng Hu: Create the .h files for card class, spell class, ritual class, player class, deck class, board class and implement player, deck, board class.

Phase 2 (Nov30 – Dec 2):

Jia Hao: Test main and player. Implement enchantment.

Chengcheng Hu: Test board and deck. Implement minion spell, ritual and TextDisplay.

Phase 3 (Dec 2 – Dec 4):

Incorporate all modules and debugs.

Write design documentation.