Report of Sorcery

Group Member:

j237sun
j58luo
j6teng

Overview:

Player: We have a player class which is our main class which implemented all the functionality on the game. First, this class has its own field such as name, life, magic, those fields represent the player's property as it mentioned at sorcery.pdf. there's also an integer indicator called player_num, which indicate which player is active during the game. Next, the Player class have fout vector of card pointer, those field indicate that the player class has a relation with card class. Here are the four vectors.

a) Board_M: A vector of minion pointer, which represent the minion part of the game interface, when we play a card from our hand, if it's type is minion, the corresponding minion card will be pushed into Board_M. when a minion is killed during the game, it will be popped from the vector and move to graveyard.

b) Hand: A vector of card pointer begin with a size of five. At the beginning of each player's turn, one card will be drawn from desk randomly and pushed into hand. The maximum size of hand vector is five, if there's already 5 card pointers inside hand, no card will be pushed. Every time a player play a card, the card will be popped and pushed into their place at board.

c) Graveyard: Whenever a minion card is killed, it will be moved into the graveyard, when a special card called *Raise Dead* is used, the latest dead minion will be popped and move back to board.

d) Deck: Deck is a vector of cards. players will draw a card from the deck randomly (card will be popped and move into hand) so long as the deck is not empty. Besides, there are four trigger indicators represented by integer, they will check some special card such as Bone Golem, Fire Elemental, Potion Seller, and Dark ritual. Three corresponding function will be used if the requirements are met. For example, the trigger_Bone Golem is initialed as 0, whenever a minion is moved into graveyard, this indicator will be set as 1, then the checkBoneGolem() function will check if there exist a minion card called Bone Golem on board, if both two requirements are met, Bone Golem's defense and attack will be reset as it said in sorcery.pdf.

Other than that, there's a player pointer that represent the opposite player inside the class. This pointer is used to access the opposite player's special field when some card are played (for example: *Unsummon*). Also, when the active player want to attack or use enchantment card to the opposite player, this pointer is also useful.

Moreover, a ritual pointer is used to represent the ritual card of a player. h_ritual (initially set as 0) is used to indicate if this player has valid a ritual card on the board, when a ritual card is played by a player, h_ritual change to 1, whenever this ritual is run out of charge or being destroyed, h_ritual will be reset as 0, this indicator is helpful when we check some cards with trigger property.

Additional, to achieve those comment in main, play_m, play_p, use_m, showhand, discard is writing since the name and life as well as magic are set private to protect our code. To access those field, those functions are needed. Also, some helper function such as "move_to_grave_yard" and "random Generate" are used to make sure that our code is clear and easy-understanding.

Card (abstract class): There are 4 subclasses of card, which represents 4 types of card. The major virtual functions of these classes are to generate and set the value of attack, defence, cost, ability of cost and charges. We also have a field to indicate whether the card have one more action chance to play. In order to achieve inspecting the enchantment of the minion, we have a specific field to store the used enchantment cards and the current enchantment cards. There are also several virtual functions to generate the type of cards, the name of cards and the cards' description. We have added several different private fields in different subclasses, but the public field of all four classes is the same.

Combining all of the above, we can generate all the basic information about cards and then achieve the most functionality of cards in player class. We also included the pointers of the opponent to activate the functionality of the opponent's cards if needed. All the data are stored in card class. According to the specific functionality of the specific card, corresponding functions in the player class will be called and run; therefore, the property of card will be achieved.

Main: We use one helper function(*updateAllAct*) to help update and check the number of action. All required commands and command line arguments are written in the main function.

Design: We use factory pattern to design all type of cards, therefore we can be easy to create a pointer to point to every kind of cards. Otherwise, we may need to create many different types of pointers to point the minions and rituals because they have some special field like defence, attack and charges. Also, according to use inheritance, we do not need to create a completely new class for each card. For example, class minion and spell can have many same methods which are different in function body, so when we can use the same ways to call the class methods for spells and minions.

Moreover, we use vector to save contents instead of using array. Sometimes, we do not know have much content we need to save in an array, so we need allow the memory from heap memory. Thus, we choose to use vector, because vector can manage the memory by itself. It is more convenient than the regular array.

Adding new functionality such as new card type is easy, as we only need to create a new subclass of card with a new public function for the type of card if needed. Since all cards have the field of name and type, we only need to mutate the new card's private field to show its property. Since the factory pattern is used, not all function must be rewritten which is time-saving. It also makes the code easy to understand.

Resilience to Change:

It’s easy to add new property and data into our game system, since we use inheritance to manage them.

Updated UML:

We delete the use of observer pattern, and add some functions in player class.

Answer to Questions:

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

To maximize the code reusability, we design a base class named Card and four subclasses named Spell, Ritual, Enchantment and Minion. For each subclass, they inheritance all virtual functions for the using of each class. For example, we put all information about Air Elemental Minion Card into the specific variables, then use the virtual function to achieve the attribute of cards. Reusability is reached.

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

Factory pattern is suitable for this situation. In *Enchantments* class, there are different functions are used to change different properties of minions, but they still have the common goal: change some variable in game. Thus, the factory pattern is suitable: functions for all possible events can be written as “a family of products with many variants”, and client can “interact with all functions through their abstract interface”.

Question 3: Suppose we found a solution to the space limitation of the current user interface and wanted to allow minions to have any number and combination of activated and triggered abilities. What design pattern might help us achieve this while maximizing code reuse?

Observer Pattern should be used since every minion can have arbitrary trigger or active abilities, thus when an ability is attached to a minion, use observer pattern to notify and change during the game.

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

Observer Pattern should be used since every minion can have arbitrary trigger or active abilities, thus when an ability is attached to a minion, use observer pattern to notify and change during the game.

Final Question:

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

A team should not have too many people, such that everyone can maximize their efficiency. The team should discuss and plan out every part of the project before splitting up the parts so that everyone understands what everyone else is doing. Also, we think we should divide the project cautiously so that every part has their specific goal and are independent with each other so there are less merge conflicts.

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

If we had the chance to start over, we would start doing the project earlier, and spend more time on this project. Since we only use factory pattern. If we start earlier, we would try to optimize our code by using more design pattern. Moreover, there is a great amount of codes in player source file and main file, thus we would find a more efficient design pattern and algorithm to realize the game system. If we have another chance, we will attempt those bonus feature. But for this time, we didn’t have much time for the bonus feature.