Design Document: Sorcery

Yiran Cao & Haochen Wu

Section I: Introduction

Sorcery is a card game based on collectible card games such as "Hearthstone: Heroes of Warcraft" and "Magic: The Gathering". The game is played on the terminal by entering commands, and it supports both text display and graphics display. Besides implementing all functionalities and cards that are outlined in the instruction document, we also add the Hero Power feature to the game.

Section II: Overview

To implement the basic functionalities of the game, we first construct the concrete Player class which contains player's basic properties (i.e. name, life, and magic), and player's deck, hand, minion slot, and the currently activated ritual card. Then, it is natural to construct the class Card and to build the aggregation relationship between the Player and Card. Since there are four different types of card, we implement each card type as a subclass of Card. Class Minion, Enchantment, Ritual, and Spell inherit basic properties (i.e. Card name, and magic cost), and basic game functionalities (i.e. Play card, use card, attack, output display) from its superclass Card. Since each card has different effects and abilities depending on its type, we then implement each card as a subclass based on its type and override the methods according to its effects.

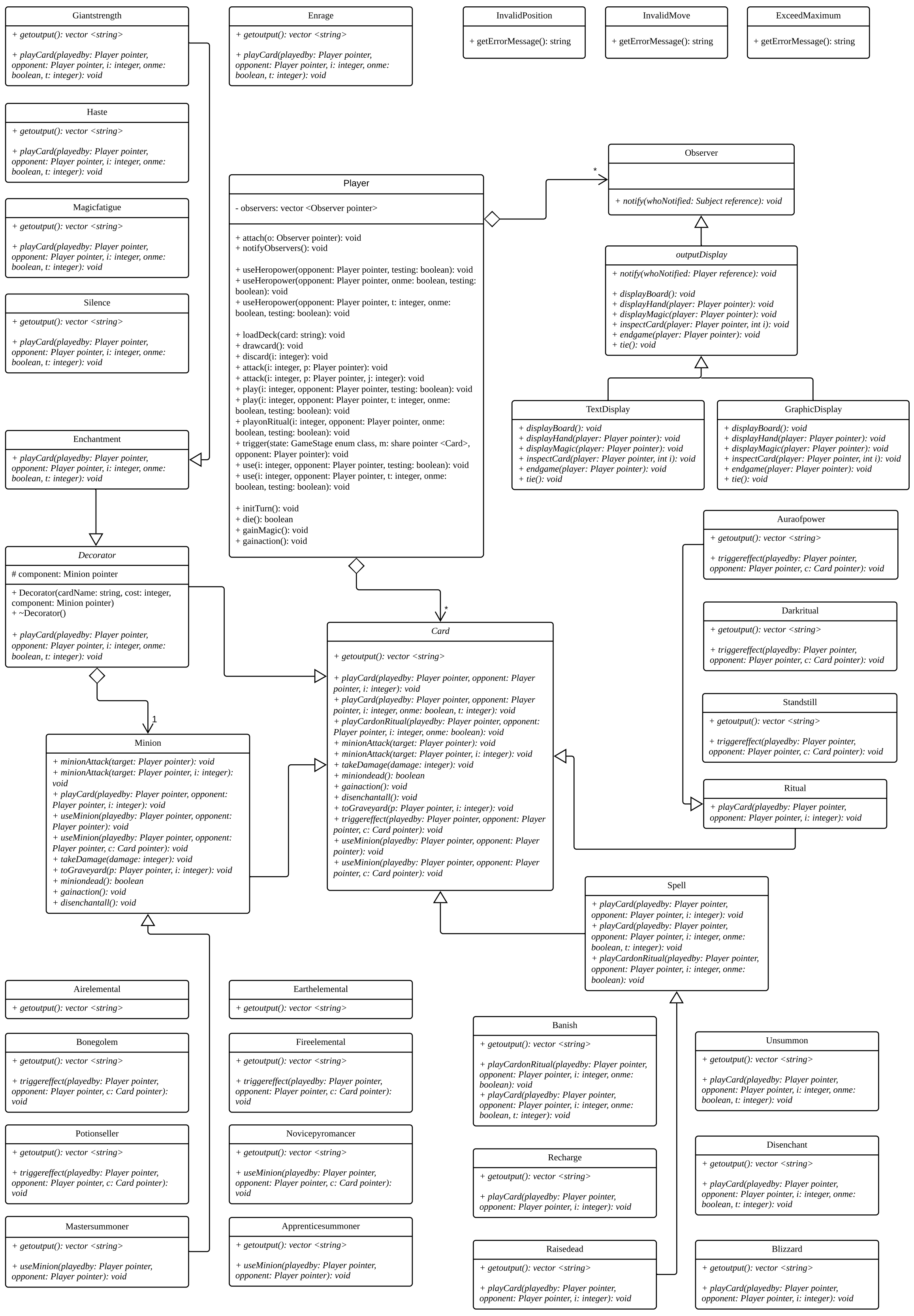
When implementing the Enchantments, we use the idea of decorator design pattern. The component is the Card class, therefore, Decorator is also a subclass of Card. The concrete components are all the concrete minion classes. Enchantment and all concrete enchantments are inherited from Decorator.

When implementing the text display and graphics display, we make use of the observer pattern. The Player acts as the subject and the outputDisplay acts as the observer.

Every time the player's status or the board changes, the player calls notify, and it will notify all outputDisplay objects. Then the displays make changes accordingly. In order to maximize code reuse, we create the abstract class outputDisplay, and two subclass TextDisplay and GraphicsDisplay which are inherited from the class outputDisplay, since they share the same fields, and the same constructor and notify method.

In addition, there are three types of exception used in our program, and each of them is created as a separate class.

Section III: UML


Giantstrength
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Enrage
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
InvalidPosition
+ getErrorMessage(): string
InvalidMove
+ getErrorMessage(): string
ExceedMaximum
+ getErrorMessage(): string
Haste
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Observer
+ notify(whoNotified: Subject reference): void
*
Player
- observers: vector <Observer pointer>
+ attach(o: Observer pointer): void
+ notifyObservers(): void
+ useHeropower(opponent: Player pointer, testing: boolean): void
+ useHeropower(opponent: Player pointer, onme: boolean, testing: boolean): void
+ useHeropower(opponent: Player pointer, t: integer, onme: boolean, testing: boolean): void
+ loadDeck(card: string): void
+ drawcard(): void
+ discard(i: integer): void
+ attack(i: integer, p: Player pointer): void
+ attack(i: integer, p: Player pointer, j: integer): void
+ play(i: integer, opponent: Player pointer, testing: boolean): void
+ play(i: integer, opponent: Player pointer, t: integer, onme: boolean, testing: boolean): void
+ playonRitual(i: integer, opponent: Player pointer, onme: boolean, testing: boolean): void
+ trigger(state: GameStage enum class, m: share pointer <Card>, opponent: Player pointer): void
+ use(i: integer, opponent: Player pointer, testing: boolean): void
+ use(i: integer, opponent: Player pointer, t: integer, onme: boolean, testing: boolean): void
+ initTurn(): void
+ die(): boolean
+ gainMagic(): void
+ gainaction(): void
Magicfatigue
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
outputDisplay
+ notify(whoNotified: Player reference): void
+ displayBoard(): void
+ displayHand(player: Player pointer): void
+ displayMagic(player: Player pointer): void
+ inspectCard(player: Player pointer, int i): void
+ endgame(player: Player pointer): void
+ tie(): void
Silence
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
TextDisplay
+ displayBoard(): void
+ displayHand(player: Player pointer): void
+ displayMagic(player: Player pointer): void
+ inspectCard(player: Player pointer, int i): void
+ endgame(player: Player pointer): void
+ tie(): void
GraphicDisplay
+ displayBoard(): void
+ displayHand(player: Player pointer): void
+ displayMagic(player: Player pointer): void
+ inspectCard(player: Player pointer, int i): void
+ endgame(player: Player pointer): void
+ tie(): void
Enchantment
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Auraofpower
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Decorator
# component: Minion pointer
+ Decorator(cardName: string, cost: integer, component: Minion pointer)
+ ~Decorator()
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
*
Card
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
+ playCardonRitual(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean): void
+ minionAttack(target: Player pointer): void
+ minionAttack(target: Player pointer, i: integer): void
+ takeDamage(damage: integer): void
+ miniondead(): boolean
+ gainaction(): void
+ disenchantall(): void
+ toGraveyard(p: Player pointer, i: integer): void
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
+ useMinion(playedby: Player pointer, opponent: Player pointer): void
+ useMinion(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Darkritual
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Standstill
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
1
Minion
+ minionAttack(target: Player pointer): void
+ minionAttack(target: Player pointer, i: integer): void
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
+ useMinion(playedby: Player pointer, opponent: Player pointer): void
+ useMinion(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
+ takeDamage(damage: integer): void
+ toGraveyard(p: Player pointer, i: integer): void
+ miniondead(): boolean
+ gainaction(): void
+ disenchantall(): void
Ritual
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
Spell
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
+ playCardonRitual(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean): void
Airelemental
+ getoutput(): vector <string>
Earthelemental
+ getoutput(): vector <string>
Banish
+ getoutput(): vector <string>
+ playCardonRitual(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean): void
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Unsummon
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Bonegolem
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Fireelemental
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Potionseller
+ getoutput(): vector <string>
+ triggereffect(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Novicepyromancer
+ getoutput(): vector <string>
+ useMinion(playedby: Player pointer, opponent: Player pointer, c: Card pointer): void
Recharge
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
Disenchant
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer, onme: boolean, t: integer): void
Mastersummoner
+ getoutput(): vector <string>
+ useMinion(playedby: Player pointer, opponent: Player pointer): void
Apprenticesummoner
+ getoutput(): vector <string>
+ useMinion(playedby: Player pointer, opponent: Player pointer): void
Raisedead
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void
Blizzard
+ getoutput(): vector <string>
+ playCard(playedby: Player pointer, opponent: Player pointer, i: integer): void

Section IV: Design

Techniques used in the project

Technique 1: Smart Pointers. Since the structure of the whole project is complicated, it will be time-consuming and error-prone to take the responsibility of memory management. Making the use of smart pointers (specifically, shared pointers) in standard library reduces our workload and avoids memory leak in a very convenient way.

Technique 2: Flags. The use of flags in main.cc helps us to make a default setting of the game and handle different options (changing the game setting), such as testing mode and the feature of hero power (to be discussed later).

Technique 3: Design patterns. We make the use of the decorator and observer design pattern. The decorator pattern is used to implement the class of enchantments. Enchantments are considered as extra feature/modifications on minions' properties, so the decorator pattern is well-fitted. The observer design pattern helps us to handle different kinds of displays (to be discussed later). We also implicitly utilized the iterator pattern as we mainly use vectors to help us handle dynamic arrays in the program. When we try to traverse the minion slot, deck, hand, or graveyard, we used range based for loops, where iterator pattern is implicitly used by vectors.

Technique 4: Exceptions. Exceptions help us to handle invalid inputs and actions. If the player enters an invalid input and or makes an invalid action, the program will throw an exception and remains the state of program unchanged. The use of exceptions provides a strong guarantee of the program.

Commenting on coupling and cohesion

We aim for high cohesion by implementing all related methods and fields inside their own class. To achieve low coupling, we use the Model-View-Controller design pattern to separate the model, view, and controller. For example, we decouple the interface and prevent the primary program class (i.e. the Card class) from interacting with the user by implementing the Player class and let the methods inside the Player class handle all inputs from the user. The Card class and its subclasses act as the controller of the game. They make changes to the program state based on the input they get. The outputDisplay class act as the view in the MVC design pattern. It works as the observer of the Player class and handles the whole display of the game separately from the other two components. Thus, each class is only responsible for one task, and this reduces the coupling between modules.

Section V: Resilience to Change

If new cards are added, we can simply create a subclass of its type, and then override the functions to implement its own effect or ability. For example, if a new minion is added, we create a subclass that is inherited from the Minion class, and we only need to override the getoutput method (each minion has a different display), and the useMinion or triggereffect method (each minion has the different activated or triggered ability). We can reuse the minionAttack function since the attack functionality is the same for all minions, and this helps to accommodate new features with minimal modifications.

If new display platforms are added, we can create a new class that is inherited from the abstract outputDisplay class. Due to the inheritance, we do not need to implement the constructor and the notify method again. We only need to override the functions for displaying board, hand, minion, and ending the game, since each platform may require a different output function. (ie. we use cout for text display, and use drawstring for graphics display) What's more, implementing all different display platforms using inheritance allows us to turn on or turn off any display easily.

If we would like to add a feature of that is similar to secrets in the Hearthstone, we can namely create a class of it and make it act like the triggered ability of the hero. Since we have implemented the triggered abilities for minions, adding such feature will not take much work and we can reuse the code to some extent.

Adding the feature of armors and health (there is no distinction between them in the current version of the game), like how they behave in the Hearthstone, requires a little more work. We need to introduce a new field that keeps track of the upper limit of health and making sure that restoring health will not bypass the upper limit while the armor has no upper limit. Additionally, taking damage should preferably reducing armors first and health later. We have a method of takeDamage, so if we revise this function, this feature can work for the whole game and other functions do not have to change.

Similarly, since we have implemented hero powers in the game, which forces us to implement the attack value for heroes, it will not be difficult to add the Weapons cards to the game as the hero has the functionality of attack now. The extra work is just designing cards of weapons and modifying the attack value of the hero when the weapons card are played.

We can see that due to our choices of design, it will not take much work if we want to add new features or cards into the game. If more were available, some features described above would be added to our game.

Section VI: Answers to Questions

Q1: How could you design activated abilities in your code to maximize code reuse?

A: Activated abilities and triggered abilities are both inherited from the class Card, which has a method called playCard with two different sets of parameters. By taking the advantages of function overloading, we saved repetitive coding to some extent.

Q2: What design pattern would be ideal for implementing enchantments? Why?

A: Decorator Pattern, because enchantments are added as additional features/modifications of minions, so we can use the decorator pattern to manage (adding and removing) enchantments that are being applied to minions.

Revised Answer for Due Date 2: In the real implementation, we do not stick to the original version of the decorator pattern, since the classic decorator pattern does not support removing, and it also makes difficult to inspect a minion. Rather, we add a new field in minion that stores all the enchantments added to the minion as a vector. In such case, if we want to disenchant this minion, we just erase the last element in this vector and make the corresponding adjustments to the minion.

Q3: 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: If this is the case, we may utilize the decorator pattern again to allow minions to have such feature. Similar to what we did with enchantments, each activated and triggered ability will be a concrete class inherited from either an abstract class called activatedAbilityDecorator or triggeredAbilityDecorator. Minion will be the Component and several specific minions will be concrete components.

Q4: How could you make supporting two (or more) interfaces at once easy while requiring minimal changes to the rest of the code?

A: Similar to what we did for assignment 4 question 5, we regard the two players as the subjects and the two observers of the subject are TextDisplay and GraphicDisplay, respectively. By overloading the output operator, we can support two interfaces at once easily.

Revised Answer for Due Date 2: In practical, we created a new class called outputDisplay, and make TextDisplay and GraphicDisplay as subclasses of outputDisplay. These subclasses share common methods inherited from the parent class, which allows us to reuse the code.

Section VII: Extra Credit Features

Feature 1: There is an additional option of "-disabletext" which allows the client to disable TextDisplay and play the game with GraphicDisplay only. It enhances the experience of playing since the command line window, will now be much cleaner. In practical, the client does not need to see two kinds of display simultaneously, so if the client prefers the graphic display, this option allows him to disable TextDisplay and enjoy the cleaner window while playing the game.

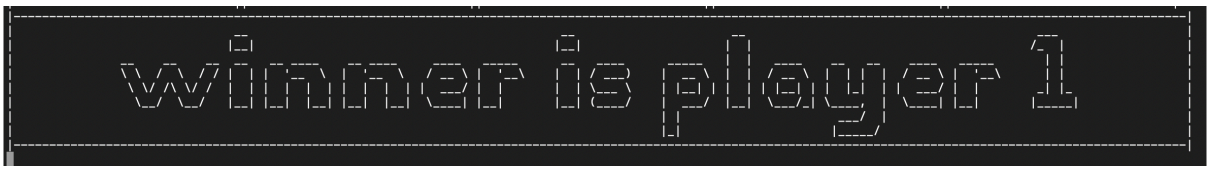
The challenging part of this extra feature is that we need to reorganize the class of different display (as you may see in the change in UML). Originally, we were simply using two subclasses to manage the display. To make GraphicDisplay and TextDisplay functional independently, we created a class called outputdisplay and make it the parent class of GraphicDisplay and TextDisplay. It took some time, but developing such feature makes it much easier if we want to add another kind of display class as they share a lot of functions from the parent class.

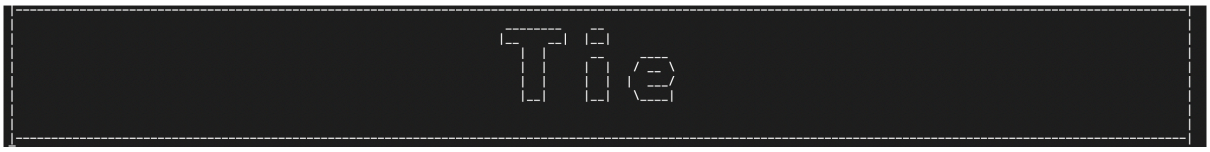
Feature 2: Hero power. Mimicking to the Heartstone, we developed an extra feature of hero power, which acts as an activated ability of the two players. There are six different heroes with their corresponding hero power available in the current version of the game. In the default mode, the heroes are randomly assigned to each player. In the testing mode, it allows the debugger to pick whichever hero he likes. With the command line argument "-disableheropower", we are also able to turn off the extended feature easily.

The challenge of adding this feature is that we had to add additional fields in the player and manage different kinds of interaction between player, minions, and rituals, and trigger abilities, similar to what we did for minions and spells. The logics are similar to handling minions and spells, but handling edge cases, especially for triggering abilities, is very time-consuming.

We could develop more hero powers if more features are added to the game, such as the upper limit of health/defense value, class of weapons, etc.

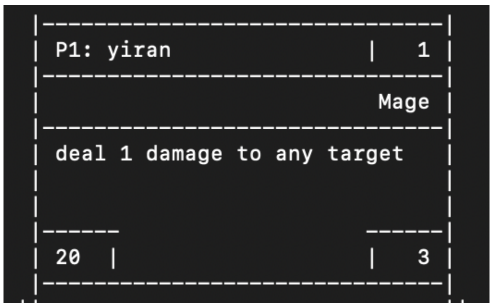
Feature 3: In order to make the output interface more interesting, we designed the text display and graphic display for showing the winner, or “tie” when the game ends. Example screen cuts are showed below.





WINNER
IS
PLAYER1

Additionally, in order to accommodate the extended hero power feature to our display, we also modified the player’s name card by using the minion’s template and display the information at the corresponding position.



```
|-------------------------------|
| P1: yiran              |   1 |
|-------------------------------|
|                          Mage |
|-------------------------------|
| deal 1 damage to any target   |
|                               |
|                               |
|------                   ------|
| 20  |                  |   3 |
|-------------------------------|
```

Section VIII: Final Questions

Q1: 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?

A: From this project, we learned that communication is very important in developing software. We were responsible for different modules, and effective communication was the key that these modules can work together. Additionally, following the schedule is very important, since a module that is to be implemented latter is likely to be built on an early implemented module. Delaying on the schedule will mess up the whole plan, and we will not likely be able to finish the project on time.

Q2: What would you have done differently if you had the chance to start over?

A: We would focus more on the overall structure first, thinking carefully about how different modules are going to be implemented, and then start the real implementation. Sometimes, we had an idea of implementation and designed the structure corresponding to that idea. However, we may encounter some challenges in the implementation later on and found that the initial idea was not working, and then we had to start over and redesign the whole structure. It costs a lot of time.