CS246 final project design document – straights

Claudia Chen

Overview

To implement the program of straights, I defined multiple classes include class Game, Player, Human, Computer, and Card.

First, class Card is a class that stores information of a single card such as rank and suit. Class Player has the method of:

```
Card(int n, char s);
int getRankNum() const;
char getSuit() const;
char getRank() const;
```

Card() is the contructor of class Card, it takes an interger as the card's rank and a char as the card's suit. Method getRankNum() returns rank of the card as an integer while getRank() returns rank as a char (e.g. if the rank is 11, getRankNum() returns 11 and getRank() returns 'J'). Method getSuit() simply returns one of 'S', 'D', 'C', and 'H' which is the suit of the card.

Then, class Player is an abstract class that stores information of each player. Its field include the player's score as an int, player's hand cards, discards, and valid plays as a vector of cards prospectively. Those fields will be updated after each play. Class Player has the method of:

```
Player();
void play(Card c);
void discard(Card c);
void dealCard(Card c);
std::vector<Card> getHandCard();
std::vector<Card> getDiscard();
std::vector<Card> getValid();
void updateValid(std::vector<Card> c);
int getScore();
void addScore(int n);
void clear();
virtual bool humanType() = 0;
```

Where Player() is the constructor of the class Player, it set the score of a new player to 0 and set all other 3 fields as a empty vector. play(Card c) removes card c from player's hand card. Discard(Card c) removes card c from hand card and add it into the list of discard. In DD1, I plan to make these two functions as virtual method and implemented them differently for Human and Computer class

which are derived classes of Player. I change my idea because I notice two types of players behaves same while play or discard a card. What's different is actually the moment of calling these 2 functions (call after receiving a command of which card to play or decide which card to play automatically) and which card to use as parameter. Those questions will be solved in main.cc instead of player.cc. dealCard(Card c) is a method that simply add card c into the player's hand cards. This method will be called when shuffle and deal the cards. getHandCard(), getDiscard(), getValid() are methods that simply return hand cards, discards and valid plays of this round as a vector of Card. updateValid() will replace the field valid with the parameter that is a vector of Card. getScore() returns the score of the player and ad Score(int n) add n to the player's current score. clear() will clear the player's hand cards, discards and valid plays then reset the score. This method is only used when restart the game when no one achieve a score of 80. Eventually, humanType() returns whether the player is a human.

Class Human don't have any new fields or methods. It only override humanType() and always return true when this function is called. Class computer has methods of setDiscard(vector<Card>) and setHand(vector<Card>). Those methods replace the player's current discard and hand cards with a new vector of card and will be used to create a new computer player that inherit the data of a human player when the human player ragequit. And class Computer will alwas return false when called humantype().
The class Game is that class that records all cards on the table. It contains the method of:

```
Game();
void play(Card c);
void shuffle(int seed,std::vector<std::shared_ptr<Player>> p);
int first(std::vector<std::shared_ptr<Player>> p);
bool validity(Card c);
void table();
void getDeck();
bool winner(std::vector<std::shared_ptr<Player>> p, std::vector<int>& g);
void clearTable();
```

In DD1, this class is also supposed to contain the 4 players as a field and all modification to Player object need to be done within a Game object. I change this because this causes a lot of method of Game are just calling methods for class Player which is very unnecessary. Therefore, class Game and class Player exist independently now. A private field of deck and method getDeck() are added just to make sure we are able to use the command deck to get all the cards on the deck. Previously, I decided to create methods of getSpades(), getClubs(), getHearts() and getDiamods() with will return ranks of all cards will corresponding suit in a increasing order. Now I combine those 4 methods into one, which is table(). This method will directly print all rank of cards on the table so that I don't need to call 4 different methods to achieve this. The play(Card c) will place card c on the table as it is played. shuffle(int seed, std::vector<std::shared_ptr<Player>> p) will shuffle current card on deck and then deal them to the players in p. The method first(std::vector<std::shared_ptr<Player>> p) is a method that decide which player is the player who has 7S and return the player's ID. validity(Card c) checks whether card c is a valid play of this

round based on current card on the table. The method winner returns true when a player wins. clearTable() clear all the cards on the table (will not change what's in the deck). This method is called when restart the game.

The class gameException is removed since it is not necessary. We don't only want the program to stop after encountering an error but also ask for the input again. This can be achieved with a boolean variable that returns true when there is an error and a loop that keep going when the Boolean variable is true.

Design

To make sure human player and computer player behaves differently but are both considered as players. I create a super class Player and then its two derived class, Human and Computer. After doing this, I can always use a vector of Player to store all players no matter the players are human or computer. Then, by creating a method that return whether the player is human or not, I can easily identify the type of the player and then decide what to do next.

In the program design, to avoid memory leak, since no vector in this program can be longer than 52 which is the total number of the cards, for all non-customized object such as int, char, and string, we add the object itself into the vector instead of its pointer. Also, smart pointer will be applied to introduce RAII to the program. A vector is used to store the player object so that it is easy to decide who is the next player after each round. The player is not stored in a queue anymore. After the game ends, we need to print the score and discards of each players in the order of player1, player2, player3 and player4. If I use queue to store the players and reordering the queue every time to get the next user, the order of player will be messed up at the end of game and the location of player1 is unpredictable. By using vector, I can still easily find the next after I figure out the player ID of the player who plays first, which means it is not necessary to use queue anymore.

With all the current functions, in main.cc, I loop for 4 times to create 4 players. Then create a new Game Object. After that, I create a Boolean win to decides whether there is a winner then start looping until win is true which indicates there is a winner. In each loop, I check whether the player is human or computer, then decides what action to take depend on the player type. This helps the game keeps going when no one type "quit".

Resiliance to Change

Since players are stored in a stack without fixed length and next player can be found with player ID of the player who play first, the number of players is not limited. More players can be added to the game after modifying the number of cards each players owns to 52/<number of players> and the program will run with no issue. What's more, if another type other than human or computer is introduced, we can just write a new derived class of Player class with corresponding virtual method

be override. My program is also resilience to change of game rules. For example, if definition of a legal play changes, the only methods I need to change is validity() since that is the only function that is used to decide whether a card is valid. After modifying this method, the game will be able to start with the new rules.

Answers to Questions

Question:
What sort of class design or design pattern should you use to structure your game classes so that changing the user interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

To allow change to game rule, I could apply factory pattern to create the card. If a new set of cards with different suit or different rank is applied to the game, we only need to modify the Creater and ConcreteCreater and then add the new suit under Game class. Also, it is possible to create an extra class textDisplay to store the user interface and each Game object should have a textDisplay object as a field. Along with the game process, textDisplay object will also be modified when modifying Game object. Finally, by creating a prtingText() method for textDisplay that print all texts needed for user interface, we can easily get out current user interface by calling printText(). With this extra classes, we only need to modify textDisplay class to change the user interface.

What's more, by create a class Player, we can change the number of players based on the rules. Also, if definition of a legal play changes, the only methods I need to change is validity() since that is the only function that is used to decide whether a card is valid.

Question:
If you want to allow computer players, in addition to human players, how might you structure your classes?
Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

To allow both computer players and human players, I create classes Human and class Computer to be the derived class of class Player. Depends on the type of the player, the player could be created in 2 ways:
std::shared_ptr<Human> p = std::make_shared<Human>();
or
std::shared_ptr<Human> p = std::make_shared<Computer>();

The biggest difference between computer player and human player is that computer select cards automatically to play or discard while human players select the card manually. Therefore, I create a humanType() method to decide whether the player is human. If yes, in main.cc, print the cards on table and read input to know what to do next. If no, automatically play the first legal play or discard the first hand card.

To make computer players have different strategy, I could add a new integer field to computer class, which each integer represents a different strategy. In main.cc, if the player is computer, the card that computer choose to play, or discard will be different depends on the current strategy of the computer. Then create a method updateStrategy(std::shared_ptr<Computer> c) under class Game which decide what strategy computer player should use depends on the current cards on the table. Every time when it is a computer player's turn, call updateStrategy() with the computer player as parameter, then the computer player will be able to change strategies as the game process change.

Question:
If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable
to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

To replace the human player, what I do is to create a new computer player and then use setDiscard() and setHand() method to make the new computer player has same discards and hand cards the human player that quit. Since I use a specific vector to store all the players that are playing the game, what I need to do now is replace the human player who quit with the new computer player. Since the replaced human player are created with smart pointer, it will be deleted automatically after the program ends and will not cause any memory leak.

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?

To write a large program, the most important part is to write a reasonable plan, so that the developer knows where to start and what direction they should follow. This part is unnecessary, the developer need to have a basic idea about how to implement each class and how to use this class to achieve some functions. Even the programming process does not start, it is still significant for the developer to be able image how the program works, so that developer will know where to start and which direction to go. Writing large program without any plan is unacceptable since all functions or objects in a large program might affect each other, adding one new function might required modification to everything that is written before if no plan was made.
Though the planning process is crucial, developers need to make sure the plan is not too detailed. It is impossible for most developers to predict everything that will happen during programming. Therefore, the plan should only provide the programming direction while all the details should be figured out during the process of writing the codes. What's more, a plan without too much details is more fault-tolerant. If part of the plan cannot be achieved, the plan should be able to accept small modification.

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

If I can start over, I will pay more attention in writing the plan of attack before DD1. With a clear plan, I can write my code more efficiently. I only pay attention to how to fulfill all the function and make the program work but does not think too much about whether the structure of the program is efficient. I do not find any good opportunity to use design pattern. Therefore, I hope I could pay more attention to the planning process so that I can avoid this issue.