CS 246 Fall 2020 Final Project - Starights

Demo

Jinyu (Leo) Chen - 20707639

1 Introduction

1.1 Game Rules

Straights is a four-player game which has the objective of getting the fewest score among the players. The game proceed with mutiple rounds, where the score is accumulated for each player at the end of each round. When a new round started, each player was randomly allocated with 13 cards from a standard 52-card deck, where the player having 7 of splades goes first (to play that card) and other players take turns to play a *legal card*. A *legal card* is one of:

1. A 7 of any suit
2. A card which has the same suit as any card on the table, and must has rank one more or one less than that card

A player must discard a card from hand if and only if there is no legal card to play - the sum of ranks of discarded cards is calculated as the score of that player at the end of each round. A round ends when all cards has been played or discarded.

1.2 Our Game Program

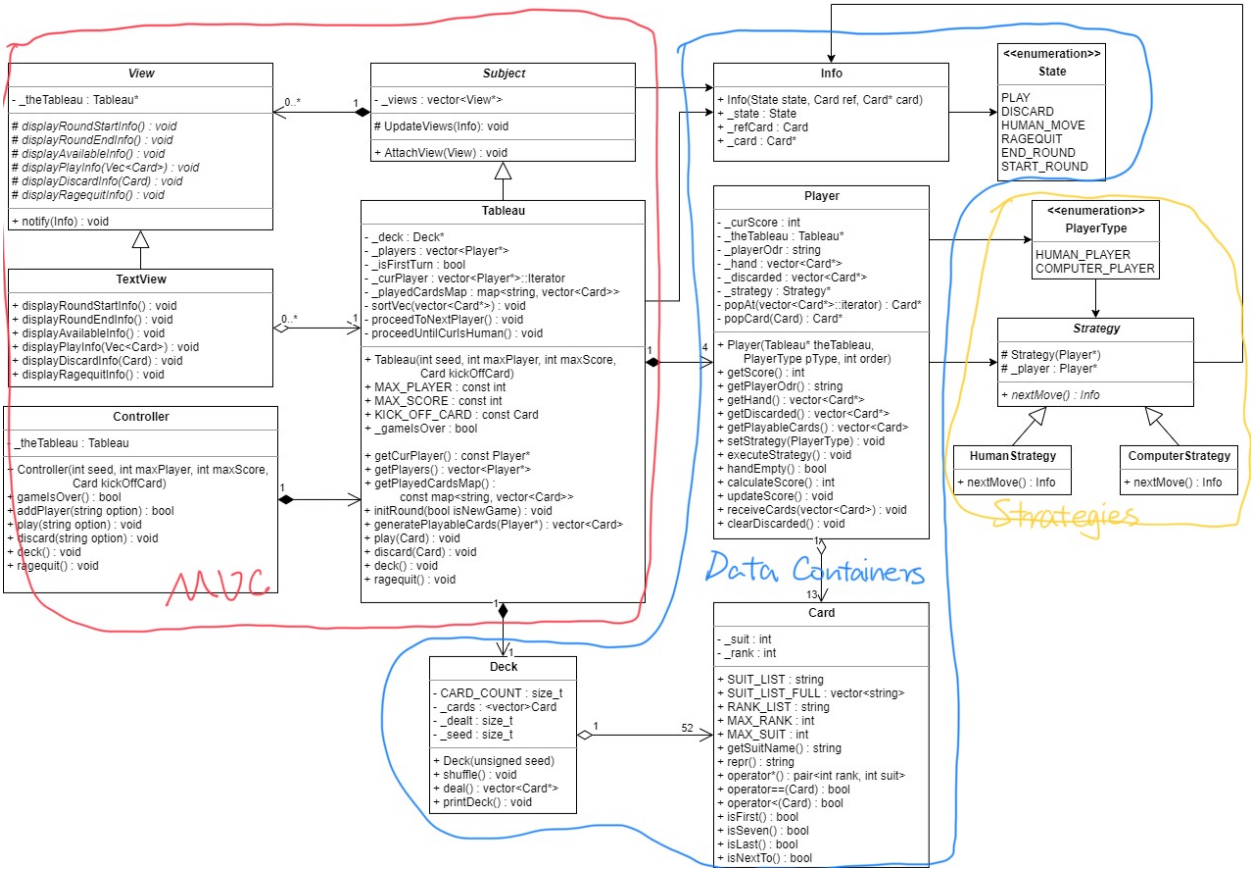
The `straights` program offers the gameplay of Straights with mutiple human players vs. mutiple computer players, or even watching four computer players finishing up a game automatically. For a move for human, the necessary information is displayed and prompt an error message if such move is invalid. The program will declare the winner(s) if the winning condition is meet.

2 Overview

Our program is mainly consist of the following parts:

- Model, View, Controller
- Data Containers
- Gameplay Strategy for Human and Computer

shown on the annotated uml:


View
- _theTableau : Tableau*
displayRoundStartInfo() : void
displayRoundEndInfo() : void
displayAvailableInfo() : void
displayPlayInfo(Vec<Card>) : void
displayDiscardInfo(Card) : void
displayRagequitInfo() : void
+ notify(Info) : void
Subject
- _views : vector<View*>
UpdateViews(Info): void
+ AttachView(View) : void
Info
+ Info(State state, Card ref, Card* card)
+ _state : State
+ _refCard : Card
+ _card : Card*
<<enumeration>>
State
PLAY
DISCARD
HUMAN_MOVE
RAGEQUIT
END_ROUND
START_ROUND
TextView
+ displayRoundStartInfo() : void
+ displayRoundEndInfo() : void
+ displayAvailableInfo() : void
+ displayPlayInfo(Vec<Card>) : void
+ displayDiscardInfo(Card) : void
+ displayRagequitInfo() : void
Tableau
- _deck : Deck*
- _players : vector<Player*>
- _isFirstTurn : bool
- _curPlayer : vector<Player*>::Iterator
- _playedCardsMap : map<string, vector<Card>>
- sortVec(vector<Card*>) : void
- proceedToNextPlayer() : void
- proceedUntilCurIsHuman() : void
+ Tableau(int seed, int maxPlayer, int maxScore, Card kickOffCard)
+ MAX_PLAYER : const int
+ MAX_SCORE : const int
+ KICK_OFF_CARD : const Card
+ _gameIsOver : bool
+ getCurPlayer() : const Player*
+ getPlayers() : vector<Player*>
+ getPlayedCardsMap() : const map<string, vector<Card>>
+ initRound(bool isNewGame) : void
+ generatePlayableCards(Player*) : vector<Card>
+ play(Card) : void
+ discard(Card) : void
+ deck() : void
+ ragequit() : void
Player
- _curScore : int
- _theTableau : Tableau*
- _playerOdr : string
- _hand : vector<Card*>
- _discarded : vector<Card*>
- _strategy : Strategy*
- popAt(vector<Card*>::iterator) : Card*
- popCard(Card) : Card*
+ Player(Tableau* theTableau, PlayerType pType, int order)
+ getScore() : int
+ getPlayerOdr() : string
+ getHand() : vector<Card*>
+ getDiscarded() : vector<Card*>
+ getPlayableCards() : vector<Card>
+ setStrategy(PlayerType) : void
+ executeStrategy() : void
+ handEmpty() : bool
+ calculateScore() : int
+ updateScore() : void
+ receiveCards(vector<Card>) : void
+ clearDiscarded() : void
<<enumeration>>
PlayerType
HUMAN_PLAYER
COMPUTER_PLAYER
Strategy
Strategy(Player*)
_player : Player*
+ nextMove() : Info
HumanStrategy
+ nextMove() : Info
ComputerStrategy
+ nextMove() : Info
Controller
- _theTableau : Tableau
+ Controller(int seed, int maxPlayer, int maxScore, Card kickOffCard)
+ gameIsOver() : bool
+ addPlayer(string option) : bool
+ play(string option) : void
+ discard(string option) : void
+ deck() : void
+ ragequit() : void
Deck
- CARD_COUNT : size_t
- _cards : <vector>Card
- _dealt : size_t
- _seed : size_t
+ Deck(unsigned seed)
+ shuffle() : void
+ deal() : vector<Card*>
+ printDeck() : void
Card
- _suit : int
- _rank : int
+ SUIT_LIST : string
+ SUIT_LIST_FULL : vector<string>
+ RANK_LIST : string
+ MAX_RANK : int
+ MAX_SUIT : int
+ getSuitName() : string
+ repr() : string
+ operator*() : pair<int rank, int suit>
+ operator==(Card) : bool
+ operator<(Card) : bool
+ isFirst() : bool
+ isSeven() : bool
+ isLast() : bool
+ isNextTo() : bool
MVC
Data Containers
Strategies

3 Design

In general, we use smart pointers on the whole design with `shared_ptr` cast to *aggregation* relationship and `unique_ptr` cast to *composition* relationship between classes.

3.1 Control Flow

3.1.1 Class Controller

This is where the control flow starts.

The `Class Controller` is designed to be, well, the controller of the game. It handles various instructions send from the `stdin` (or a file) and according to the command (possibly with options), call the methods in the *Model*, which is the `Class Tableau`. Note that `Class Controller` is **composed** of (i.e. owns) `Class Tableau`, so `unique_ptr` applies.

3.1.2 Class Subject & Class Tableau

This is the next spot for the control flow.

The `Class Tableau`is designed to be `Model` part of MVC design pattern. It acts like a *dealer* of the card game, registering and manages:

- each player and a deck of cards (`_players`, `_deck` fields) and their action
- the setting of the game (`MAX_PLAYER`, `MAX_SCORE`, `KICK_OFF_CARD` fields)
- the recording of any played cards (`_playedCardsMap`)
- the current play in turn (`_curPlayer`)

I must mention that the core algorithm of controlling the game is implemented by interacting these fields and methods `_curPlayer`, `proceedUntilCurIsHuman()`, and `initRound(bool isNewGame)`:

- initialize a new round according to `isNewGame` by `initRound(bool isNewGame)`;
- proceed the game by `proceedUntilCurIsHuman()` and update `_curPlayer`;
- until a win or inside `proceedUntilCurIsHuman()`, start another round by `initRound(bool isNewGame)`

In this way, the two methods call each other and stop when a human player is meet and awaits instructions from `Class Controller` or the game ends.

This class inherits from `class Subject`, which forms a *subject* corresponds to *observers*, the `Class View`. It collects informations (`Class Info`) for each play and the entire game for presenting and send it to *observers* by calling `UpdateViews(Info)`.

3.1.3 Class View & Class TextView

This is the desination for the control flow.

All information is displayed by six overriden methods in `Class TextView`, driven by the horse function `notify(Info)` which decodes passed `Class Info` package.

These classes are owned by `Class Tableau`.

3.2 Data Structure, Algorithms, and Information Stream

3.2.1 Class Player

This is the main part where the information about Cards (`_hand`, `_discarded`), scores (`_curScore`) stores. Each player is given a shared ownership of 13 instances from `Class Card`.

3.2.1.1 Class Card This is where a card specified. It supports reading, comparing, and positioning a card object by overriding opertors or providing public methods.

3.2.1.2 Class Deck `Class Deck` is the owner all cards objects from `Class Card`. It supports shuffling all registered cards inside and dealing them to each player.

Back to Class Player: Most importantly, a strategy (`_strategy`) is bond to a player which implements an algorithm for generating an move, which is a pointer to the `Class Strategy`.

3.2.2 Class Strategy and derived Class HumanStrategy, Class ComputerStrategy

This is the other core part of the program. It (the base Class Strategy) has the effect of modifying internal state of `Class Player` and also, send the information stream by returning a `Class Info` package to the `Class Tableau`. In such way, `Class Tableau` knows what happend and could decide when to stop the game and transmit the information stream to update the views (`Class TextView`).

3.2.2.1 Class Info, enum Class State, enum Class PlayerType They together bulids up an information package to be send.

4 Resilience to Change

I would like to descibe possible changes to the program, from more specific, trivial ones to more abstract, substantial ones.

4.1 Game Winning Conditions

What if we want to invite more players into the game? Change the **wildcard** *(Card with Rank 7), or the* **kick-off card** *(Spade 7)? Want to play longer time?*

They can be easily achieved by changing fields `maxPlayer`, `kickOffCard`, and `maxScore`, which are arguments for constructing a `Class Controller`. So that it sets up the corresponding fields in `Class Tableau`.

4.2 Different Cards

Wanna play the weird cards brought in an one-dollar store? Define your own cards?

They can be easily achieved by modifying fields `SUIT_LIST`, `SUIT_LIST_FULL`, `RANK_LIST`, `MAX_RANK`, `MAX_SUIT` in `Class Card`. Notice that the owner of it, `Class Deck` will construct a deck of cards according to these fields.

4.3 More Difficult Computer Players

The game is too easy for you?

The **Strategy Design Pattern** is included for this. We can easily derive concrete classes out of `Class Strategy` by overriding the abstract method `nextMove()` for different levels of difficulties.

4.4 House Rules

Wanna play the game your own way?

Thy can be not so esaily achieved by modifying the algorithm-related methods inside `Class Tableau`, `Class Player`, `Class Strategy`. Luckily, the program is designed to achieve **high cohesion and low coupling**:

- many get and set methods are implemented
- isolation between control flow and information stream
- run-time polymorphism design in Strategy: `Class Player` does not know if him/herself is a Human or Computer, the specific strategy used is determined during the game
- data structures are easy to use, which provides representation strings and comparison operators.

4.5 GUI Views

The text view is out of style?

Thanks to the **MVC Design Pattern** designed in the program. A new view can be easily implemented as long as it overrides the needed six display methods inside `Class View` and attached to the observer list in `Class Subject`.

5 Answer to the Questions in Project

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

Answer: Please see More Difficult Computer Players , House Rules and GUI Views. Basically all of the previous part.

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?

Answer: Please see More Difficult Computer Players.

Qustion: 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?

Answer: Please see More Difficult Computer Players and the third point in House Rules.

6 Extra Credit Features

I really would like to implement more features, but all my finals were accumulated around 16th...

I do implement the whole program using smart pointers (without any `delete`).

7 Final Questions

Qustion: 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?

Answer: I learned that, for large programs, it's super important to seperate the control flow and information stream (this is hard...). I wasn't aware of this until when I was debuging - *who modified this part of data? who should be responsibly for that part of data?* Finally I managed to seperate the data and package the information stream into a `Class Info` to be transmitted. This works fine.

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

Answer:

- Start and examine the time cost of a large project earlier
- Have a clear mind in designing classes so that the control flow and information stream are seperated
- Get a better chair. Sitting on it writting code for three continuous day is a plain.

8 Conclusion

People invented (or rather discovered?) **object-oriented programming** for better modelling the real-world. I feel this deeply when redesigning my classes - *what API should this class provide? what data it should owns?* This can actually be solve by **common-sense**: to play a card game, we (usually) need a *host* (`Class Tableau`), a *deck* of *cards* (`Class Deck`, `Class Card`), and, well, *players*(`Class Player`). And players have their own *strategies*... This is probably the mind behind the mysterious object-oriented programming.

I really did learned a lot in this project and everything learned in class comes in handy. Thanks for the term!!