CS246 Final Project Straights : design.pdf
By Mark Liu
Fall 2021

1. Overview

I made several minor changes to my UML from DD1 to DD2. These changes mainly consisted of changes in accessibility, parameters, return values, and three new methods in the Display class to display messages for certain scenarios I did not account for before, such as when a discard is invalid. I also added a new DropLowestStrategy child class of Strategy for one of my enhancements. More information regarding this is enhancement is available in the Extra Credit Features.

Below is an overview of each class and how they communicate with each other:

`Card:`

Card is a concrete class that contains all the attributes and methods that a card in a deck should have. It has a rank, suite, and a method to convert the rank into a number (Jack=11, Queen=12, etc). It also has an overload operator to check if 2 Card objects are equal. Card objects are equal when they have the same rank and same suite.

`Display:`

Display is an abstract parent class that contains all the attributes and methods required to display all the different messages in Straights. For example, displayInformationForTurn, displayEndOfRoundMsg, displayWinner, and many more. These methods are pure virtual are implemented in the concrete child class ConsoleDisplay. High cohesion and low coupling is achieved with Display and ConsoleDisplay. Low coupling is achieved since the display methods are called via function calls with basic parameters, such as a player id or vector of Card objects. All the methods inside Display (and its children) are only used to display the different messages that are required for Straights. They do not handle any logic regarding when to display the messages.

`straights.cc:`

straights.cc contains the main method that accepts an optional seed from the user and instantiates an instance of the NormalGame class. It calls the executeGame() method of NormalGame.

`Game:`

Game is an an abstract parent class that contains all the attributes and methods to run the game of Straights. It has a concrete child class called NormalGame that implements its pure virtual methods. This class handles everything that has to do with the actual game logic and the order of the steps.

For example, Game contains: the deck using a vector of Card objects, the cards on the table using a vector of Card objects, the seed for shuffling the deck of Card objects, vector of shared pointers for each player, the display (and calls all the required methods), method to start the game, method to start each round, method to get the starting player of the round, method to check if the round should end (that is, the cards ran out), method to check if the game should end (that is, a player reached >= 80 points).
The concrete child classes of Game (such as NormalGame) must implement a method to get the current legal plays and the logic to handle user's actions (play, discard, deck, etc.). These are implemented in concrete children classes because they are likely to change if game rules change.

`Player:`

Player is an abstract parent class that contains all the attributes and methods to handle the logic of a Straights player.
For example, Player tracks their id, their new score gained during each round, and their total score. It stores their hand and discarded cards using a vector of Card objects for each.
The concrete child classes of Player (such as HumanPlayer and ComputerPlayer) must implement the getAction pure virtual method that returns that the player's action for their turn. The currently supported actions are: playing a card, discarding a card, deck command to show deck, quit to quit and end the game, and ragequit to allow a ComputerPlayer to take over. Any other command will display a message to indicate that it is invalid.
The HumanPlayer concrete child class implements the pure virtual getAction method by allowing the user to input the command using standard input.
The Concrete child class ComputerPlayer implements the pure virtual getAction method by calling the executeStrategy method of a concrete child class of Strategy, such as NormalStrategy. See below for more information on Strategy.

`Strategy:`

Strategy is an abstract parent class that handles the strategy for making moves during the ComputerPlayer's turn. Concrete children of Strategy must implement the executeStrategy pure virtual method that returns what action the ComputerPlayer should play given the current legal plays and hand of Card objects.

2. Design

Below is a description of the techniques used to solve various design challenges.

i) Template Method Design Pattern:

The steps required to execute the game of Straights will likely not change. For example, each game of Straights will follow the same algorithm:

```
Create 4 players
While game is not finished:
    Execute a new round
    After the round finishes, determine if the game should finish
    If someone won, display the winner(s) and end the game
```

Similarly, each round will follow the same algorithm:

```
Shuffle the deck
Deal the cards
Find the starting player
While the cards did not run out:
    For each player
        Get their action
        Handle their action
Update each player's total score
Clear the table
```

The order of steps in the round will not change but the way that the player's action is handled may change due to a rule change, change in existing features, or if new features are added. Thus, the Template method design pattern was implemented to accommodate this possibility. In the template method design pattern, there is an abstract parent class. The method that contains the algorithm is not virtual; the steps and their order may not be changed. The concrete child classes implement the steps in the algorithm.

In this case, Game is the abstract parent class with the algorithm executeGame and executeRound. NormalGame is the concrete child class that implements the step handleAction (where change in behaviour can happen) in the algorithm.

ii) Strategy Design Pattern:

The strategy that the ComputerPlayer uses to make their move each turn should be able to adapt and change in the middle of the game. To accomplish this, I implemented the Strategy design pattern. In the strategy design pattern, there is an abstract parent class with an abstract algorithm interface. The concrete child classes implement this algorithm in different ways.

I implemented this using Strategy as the abstract parent class with executeStrategy as the pure virtual algorithm. This algorithm returns the move that the ComputerPlayer will make given their current cards in hand. I created the concrete child class NormalStrategy to implement executeStrategy to play the first legal card, or discard the first card in hand

if there are no legal cards. DropLowestStrategy implements executeStrategy by playing the first legal card, or discard the card with the lowest rank if there are no legal cards.

3. Resilience to Change

I designed my project so that adding new features and making changes to existing features, rules, or behaviour, would as little modification to the original program and as minimal recompilation as possible. Below are various specific ways that the Straights game can change and how my project can accommodate these changes.

If a new display is required:

If a new display is required instead of printing to standard output, my project can accommodate this by creating a new concrete child class of Display that implements all the virtual display methods. Additionally, the high cohesion and low coupling of Display aforementioned makes it easy to port this display functionality to another game with different game logic, as long as that other game requires the same messages.

If a new type of player is required:

Potential new player types can include a HybridPlayer that alternates between turns being made with the computer versus being made with manual human commands, HumanFirstPlayer that is equivalent to the existing HumanPlayer but switches to a ComputerPlayer if the human takes too long to make a move, and many more. These new player types can be implemented by simply creating a new concrete child class of Player that implements the getAction method.

If there is a change in existing syntax:

Existing command syntax can change. For example, “play 7H” can turn into “PLAY 7H,” “play seven of hearts,” “play H7,” or any other variation. Other examples include: “discard 7H” turning to “discard card: 7Hearts” and “ragequit” turning into “RAGEQUIT!!!”

All of these changes in can easily be made by changing the implementation of getAction inside HumanPlayer class.

If there is a change in commands:

There can be new commands. For example, commands can be added to: display the player’s own discards, display the player’s own points, display the current round, display the current table, display who currently has the fewest points, display who currently has the most points, etc.

All of these changes can easily be made by changing the implementation of getAction inside HumanPlayer class and changing the implementation of handleAction inside NormalGame class.

If there are new cards:

New cards such as Jokers can be added and the game should adapt accordingly. To accommodate this change, a new concrete child class of Strategy and a new concrete child class of Game are required. The new concrete child class of Strategy needs to implement a executeStrategy to provide a new strategy for the ComputerPlayer to handle when it is given the new cards. The new concrete child class of Game needs to implement the handleAction method to handle the new cards being played.

If there is a new strategy or change of strategy during the game:

The strategy used by the ComputerPlayer to make a move during their turn can change. To accommodate this change, a new concrete child class of Strategy is required that implements the executeStrategy method.

If there is a change in legal cards:

The rules that determine which cards are legal to play can change. To accommodate this change, a new concrete child class of Game that implements the getLegalPlays method is required.

If the Straights game rules change:

The game rules of Straights can change. For example, increase player points when a card is played instead of when a card is discarded, the increase in player points is a constant amount rather than being based on the discarded card rank, player's discarded cards are not kept track of, ragequit wipe a player's points and let another human take over instead of letting the computer take over, along with many more ways that the game can change.

All of these can be implemented by just creating a new concrete child class of the Game abstract parent class that implements the handleAction method.

To ensure minimal recompilation is required, I utilized forward declaration in all the header files so that the implementation of various methods can be changed while keeping the header file unchanged. This way, the implementation files that #include those header files do not need to be recompiled. Existing files will also have minimal change due to the use of abstract parent classes and abstract methods that allow concrete child classes to implement them.

4. 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 from text-based to graphical, or changing the game rules, would have as little impact on the code as possible? Explain how your classes fit this framework

To change the user interface from text-based to graphical, a new child class of "Display" is required. It needs to implement all of the abstract methods such as "displayHorCPrompt()," "displayBegOfRoundMsg()," etc.
To change the game rules, for example, including the Joker cards, a new child class of "Strategy" and a new child class of "Game" are required. The new child class of "Strategy" needs to implement a new strategy for the "ComputerPlayer" according to these new rules. The new child class of "Game" needs to implement "handleAction" to handle moves according to the new rules.

Question: 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 class structures?

To allow the computer to change strategies as the game progresses, a new child class of "Strategy" is required. For example, one can add "PassiveStrategy" or "AggressiveStrategy" child classes. These new child classes need to implement the "executeStrategy" method to decide what moves the "ComputerPlayer" should make based on the strategy.

Question: How would your design change, if at all, if the two Jokers in a deck were added to the game as wildcards i.e. the player in possession of a Joker could choose it to take the place of any card in the game except the 7S?

To handle the new behaviour of including the Joker cards, a new child class of "Strategy" and a new child class of "Game" are required. The new child class of "Strategy" needs to implement a new strategy for the "ComputerPlayer" according to the rules for playing the Joker. The new child class of "Game" needs to implement "handleAction" to handle the Joker card being played.

5. Extra Credit Features

- I have implemented three enhancement features. The first one is the ability to start a game from a saved game state. The second is a new strategy called the 'Drop Lowest' Strategy that the Computer Player can use when it's their turn. The third is not explicitly managing memory.
- **i) Start Game From Saved State:**
    - What I did:

By passing the word ‘loadMode’ command line argument after the seed, the game can read data from an input file containing the saved state of a game such as cards currently played on the table, each player’s score, the cards each player has, the discarded cards each player has, the ID of the current player’s score, and the moves that the player’s will make. This data will be used to initialize the game of Straights.

I implemented this by checking the command line arguments in straights.cc. If ‘loadMode’ is one of the arguments, we read in all of the the game state data and execute a new game starting with that data.

- Why it was challenging:

  It was challenging keeping track of all the things that should change depending on if the game should start “fresh” or start from the saved state. For example, if the game starts from a saved state, we should not deal 13 cards to each player anymore, instead we set each player’s hand to their hand of cards read in from the input file. Also, if the game starts from a saved state, only the first round should start using that data; the following rounds should start “fresh” again.

- How I solved it:

  I solved this problem by inspecting the flow of the game logic and identifying each section that is specific to a round starting from “fresh” versus starting from a saved state. Once these section were identified, it was only a matter of adding if-statements to check if we are in ‘loadMode’ or not. Following this, I heavily tested the game to make sure that the game states load correctly and if the ‘loadMode’ flag was not passed as a command line argument, then the game runs like normal. More details on the format of the saved game state file is available in demo.pdf

- **ii) ‘Drop Lowest’ Strategy:**
  - What I did:

    By passing the word ‘changeStrat’ command line argument after the seed, the game will prompt the user to input what type of Strategy the Computer Player should use. The default Strategy is called the NormalStrategy which tells the Computer Player to play the first legal play in hand, and if there are no legal plays, discard the first card in hand. The ‘Drop Lowest’ Strategy functions similarly but will instead tell the Computer Player to discard the card with the lowest rank when there no legal plays.

  - Why it was challenging:

    There were hiccups in implementing the ‘Drop Lowest’ strategy as the Computer Player did not know which strategy to use. During the construction of the Computer Player, I needed to set the strategy that it was going to use for each turn (NormalStrategy or DropLowestStrategy).

  - How I solved it:

- To solve this, I edited the Computer Player constructor to add an integer to represent the strategy that the player should use. I added a default value of 1 for the strategy in the Computer Player constructor so the code I wrote before adding this DropLowestStrategy would still work since 1 represents the NormalStrategy.

- **iii) No Explicit Memory Management:**
  - What I did:

    I did not use the 'new' keyword to allocate any heap-allocated memory and did not use 'delete' to free any memory.
  - Why it was challenging:

    It was challenging to figure out which classes should create new pointers and which classes should not. With my class hierarchy, I wanted to make sure that classes only had access to the absolute minimum number of pointers that were required and I also needed to make sure everything was freed correctly when the game ended. This includes both the case where a player wins the game and it ends and the case where a player quits and the game ends.
  - How I solved it:

    I used smart pointers in the Game class to create new Players and to create the Display which is responsible for displaying all the messages of the game. Since the Game class is responsible for handling all the game logic, it made sense to create them here. When other classes such as Display required access to these pointers, the Game class would then be able to pass them as arguments. When the game ends, all the pointers were freed correctly.

6. Final Questions

Question a) 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?

I worked alone and a lesson I learned is that it is very important to plan everything out ahead of time. While I was creating my UML diagram and designing the way my classes would interact each other, I saw various posts on Piazza asking questions regarding implementation. Because of this, I panicked a little bit because I thought I was behind since it seemed like everyone else had already started programming the project. However, I'm glad that I didn't get pressured and rush the planning process to jump into coding. Because I spent a large amount of time planning out the UML diagram, once it was time to start programming, I rarely ran into any issues and was able to program all the main functionality of Straights in a relatively short amount of time. I didn't run into any design problems mid-way that would require me to overhaul code and make drastic changes to

what I had. I belive that if I didn't spend so long planning out the project, that definitely would have been the case.

Question b) What would you have done differently if you had the chance to start over?

If I had to chance to start over, I would have started earlier and implemented a visual display. This visual display would display all the same information and messages as the existing ConsoleDisplay but could do so using something similar to XWindows. This would have made the game more visually appealing, polished, and fun to play. Because I started reasonably early, I had time to program and test everything, but if I had started even earlier, I would have been able to develop more extra features such as this visual display.

7. Conclusion

Thank you for marking this project, this course has been one of my favourite :)