CS246 Final Project – Design

Introduction

The final project that I chose to do was the individual assignment, specifically the card game Straights.

Overview

My project is made up of eight classes: Game, Player, Human, Computer, Deck, Card, Suit, and Rank. It is structured so that most of the actual game (straights) is executed inside of game.cc, but uses aspects from other classes to build the game. Most of the game set up is done inside main.

I will start with the two most simple classes I have in my program: Suit and Rank. These are two enumeration classes that list out all of the card ranks used in the game (ranging from Ace to King) and all of the suits (Clubs, Diamonds, Hearts, Spades). I used an enumeration class for readability and to compare ranks/suits throughout the game.

The next class I will cover is the Card class. This class is very simple as well since it is just made up of two variables: card_rank and card_suit. Since these are private variables, the Card constructor sets the rank and suit (which do not change throughout the game), I have implemented getter functions, and lastly a print_card function which just prints out the card in the specified format described in the project description.

The Deck class is used to keep track of the deck used throughout the game, as well as the cards displayed on the table. Deck is comprised of two vectors of Card pointers, initial_shuffle and playing_deck. Initial shuffle is only used by the main deck that is used throughout the game (not the Clubs, Diamonds, Hearts, and Spades deck that are used to display cards on the table). There are many methods in the Deck class.

- The shuffle method shuffles the playing deck randomly based off the seed or a random number generated
- The deal method deals out cards to all the different players
- The restore method sets the playing deck to equal the initial shuffle (so that at the end of the round the deck is set back to how it was before starting a new round)
- The display_deck method prints out all the cards in the deck in the way described in the project decription
- The display_line method prints out all the cards in the deck in a single line instead of 13 per row. This is only used on the Clubs. Diamonds, Hearts, and Spades decks when displaying the cards on the table.
- clear_playing_deck clears the playing deck
- Other setters are used to set the initial shuffle and playing deck vectors

The Player class is used to keep track of each individual players identity (whether they're a human or computer player), score, hand, and discarded cards. This class is a superclass to my Human and Computer classes since both have many of the same features, with only a difference

in the implementation of the play and discard methods. For this reason, play and discard are virtual inside of the Player class, and will be discussed later. Other than getters and setters, the Player class has the following methods:

- clear_discarded is used to clear the vector of discarded cards. This is done at the end of each round after calculating the score so that the player begins the next round with a full hand and no discarded cards.
- print_cards is used to print out the players given cards, whether that be the hand vector of Card pointers or the discarded vector of Card pointers. This is used when it's a human players turn (their hand gets printed out) and at the end of a round when each player has their discarded cards displayed.
- calc_score calculates the new score for the player using the integer value of card rank (by static casting) and summing the players entire discarded vector of Card pointers.

With Player as the superclass, I have two subclasses: Human and Computer. These two classes inherit the variables from Player (identity, score, hand, discarded) as well as all the other methods, except play and discard which get overridden.

Inside of human.cc, I don't implement the play method because all of the commands and details are handled inside of game.cc for simplicity (otherwise I would have had to transfer lots of unimportant data over to the play method which overcomplicated things). However, I do implement the discard method which takes in a given card (the one input by the player), adds it to the discarded vector, then removes it from the players' hand.

Inside of computer.cc, I implement both the play and discard methods. The play method takes in a vector of Card pointers, legal_moves, and then plays the first legal move in that vector. After printing it to the screen, it then removes that card from the computer players hand. The discard method takes in a single card (the first card in the computer players hand), places it at the back of the discarded vector, then removes in from the players hand.

The last class, and by far the most complex, is the Game class. The game class is comprised of a vector of Player pointers (players), as well as five Decks (playing_deck, clubs, diamonds, hearts, spades). Since most of the actual gameplay happens here, each of the following methods is significantly longer in game.cc than any other class' methods.

- The score method takes in the vector of Player pointers and calculates each players score, modifying the players' score variable. It outputs the results as well. If any players' score surpasses 80, the game ends. This method also determines the winner of the game (and if there are more than one winner!), then returns a Boolean depending if the game is now over or not (true if it is, false if it isn't).
- The play_game method is the game that actually runs the game. It uses two while loops (one nested in the other) where the outer loop cycles through the rounds and the inner loop cycles through each players turn during the round.
- The command method is used only by human players, and acts as the Human "play" method. It takes in the players' turn, all the legal moves the player can do, and a vector called loose_cards that are all the new cards created when either playing or discarding a card (these new cards are not a part of the initial playing deck, which gets created and deleted in main, so they are deleted at the end of the game after however many rounds have been finished). This method takes in the user input and determines what command was entered, then performs whatever action is required by

the command (by either calling other functions or completing what needs to be done in the if statement).
- The display table method displays all of the cards on the table during a human players turn (Clubs, Diamonds, Hearts, Spades).
- The convert method converts a human player to a computer player, and returns that new human player to replace the old players spot (while deleting the old player).

Lastly, my program is structured so that most of the setup begins in main. In main, I create all of the card objects, the main playing deck, the players, the table decks (Clubs, Diamonds, Hearts, Spades), and straights itself. All of this data is then sent to game.cc where the majority of the gameplay happens. After the game is over, I delete all of the player objects, card objects, decks, and the game object.

Design

Being the first large project I've ever completed, I had to implement several different techniques to solve the various design challenges.

Firstly, to have high cohesion I wanted to use object oriented design principles such that every class is closely related. I did this by designing my classes in a way where each class has its own unique purpose. My Card class is simply made up of two variables for the rank and suit. My Deck class uses the Card class to create its own vector of Card pointers, which point to Card objects. All of the methods within my Deck class are related to the Deck itself: shuffling, dealing, displaying, etc. My Game class is also designed similarly, where all the game rules and features are implemented into the class. This includes calculating the score, determining if moves are legal, converting computer players to human players, and more. In each separate class, my elements pass data to each other and cooperate to perform exactly one task. That task could be to keep track of all the player information, or to perform all the functionalities one would like to on a Card object.

The next main goal I had while designing my program was to ensure that there was low coupling between modules. With low coupling, should I have to make a change in one class, I do not have to make many changes elsewhere. My program has low coupling since my modules only communicate via function calls with basic parameters/results, and my modules at most only pass vectors back and forth. They do not share any global data nor have access to each other's implementations. For example, when I had to change how to calculate the scoring for each Player, I only had to modify the calc_score function inside of player.cc, which did not affect any of my other modules. If there is a rule change, for example such that computer players discard their last card in their hand and play their last legal move, I would only have to modify my computer.cc file to adjust for this change. Since I implemented a design that has low coupling, this ensures that any necessary changes could be made without having to modify vast amounts of code.

This leads into another important design concept that I implemented: Encapsulation. Every object in my program is encapsulated so that all the implementation details are sealed away. The only way to modify or access any data inside of my objects is to use the provided getter or setter methods. To do this, I made sure to set the access modifier to private for every variable or vector I created. All my classes have private data fields, except Player which uses

protected (since the subclasses Human and Computer must be able to access the superclass' fields). This leads to my next design choice, which was to use Inheritance.

The next step was to design my Human and Computer classes. Although different in how each type of player makes its moves or discards its cards, they're identical in terms of required variables and methods. Instead of creating two separate but nearly identical classes, I instead decided to make use of Inheritance by creating a Player superclass, with Human and Computer as its subclasses. This way I didn't have to make identical methods but just had to override the few that needed to differ between the two classes. In my program, a Human Is-A Player and a Computer Is-A Player. So, the four variables in my Player class (identity, score, hand, discarded) alongside all the methods are inherited by these subclasses. Since I did need some differences between the two classes, specifically with the play and discard methods, I made them pure virtual in the superclass and overrode them in the subclasses.

While I stuck pretty closely to my initial UML diagram, I did make some changes; mainly, I had to create a separate Card and Deck class. I realized very early on while coding that having a deck inside of my card class would overcomplicate things, and would go against the high cohesion goal I was aiming for, so I made modifications. I also added extra methods that I did not initially plan on having / did not have the foresight of needing.

Lastly, I will mention a few challenges I ran into and how I overcame them through the design of my program. I ran into an issue when attempting to create my Card objects using Rank and Suit. Since I sometimes need the numeric value of the rank, and other times I need the actual Rank, I used static casting to work around this. I would cast an int into a Rank or Suit data type, or vice versa where I would cast a Rank into an int. Next, one of my biggest issues was memory management. I had a lot of memory leaks after running my code through valgrind, and realized that I would need to ensure I'm only creating objects when absolutely necessary, and deleting them immediately when I no longer need them. To do this I made only one deck that gets used and passed around throughout the entire game (the main deck). On human players' turns, I had to create new Card objects too, but I kept track of these cards in a separate vector so that they can be cleaned up at the end. By implementing these two techniques I was able to ensure that my program leaked no memory.

Resilience to Change

My program is able to adapt to new changes with only a few small changes throughout the code since I made use of object oriented design principles when designing my program. Each class is highly cohesive since the variables and methods of a class relate specifically to the purpose of that class. Each class is intended to serve one exact purpose. The reason that this would support the possibility of various changes is because I can easily implement new classes or modify existing ones depending on what is required. For example, if I was asked to add a new rule where if a player doesn't have to discard an entire round they get a -10 subtraction from their current total score, almost no code would have to be modified; I'd simply have to add an if statement inside of my score method inside of game.cc. If, on the other hand, I was asked to implement an entirely new feature, say a new type of player that alternate between computer and human each turn, I could easily create a new class specifically for this feature that inherits from my Player class. Due to the high cohesion of my classes, the adaptability of my program is at a level where very few changes would need to be made to other classes should any of the program specifications change.

Another reason why my program can support various possible changes is because there is low coupling between modules. This design principle goes hand-in-hand with high cohesion. Each of my classes relies very little on other classes so that modifications of one class do not affect the others. To do this, I ensured that my modules communicated only through functions calls with basic parameters, and passed only vectors back and forth. For example, my Player class (and by extension Human and Computer classes) do not need any information from my Game or Deck classes. Therefore, if I need to make modifications to my Player class, no other area would need to be modified. Another example is my Deck class. Only one function (deal) relies on a vector of Player pointers, and the implementation of deal would not even be affected by any modifications made to the Player class. Since I structured my program in this way, should I make any changes to any classes, very few to no changes would be required in alternative classes. For this reason my program is very adaptable to any changes that would be necessary.

Answers to Questions

Question 1: No change to answer. If the game rules changed, I would be able to modify my Game class and depending on how significant the rule change is, that would be the only class that would need a change.

Question 2: If I were to have different types of computer players that use different strategies, I would still choose to implement the Strategy design pattern. I would apply this pattern to my Computer subclass, where different instances of Computer will have different strategies applied to them.

Question 3: My design would not need to change if two Jokers were added to the game as wildcards. All I would need to do is modify main to add two Jokers to the playing deck, and add one game rule in my games.cc to account for the Jokers wildcard ability. The clean-up of cards and creating a game object would not need to change.

Final Questions

(a) I learned that writing large programs is very difficult. It took a long time to decide how to design the program itself, and the best way to implement it. Carefully planning out each class and how they interact with one another and can be used effectively without relying too heavily on each other shortened my workload, but I still need lots of practice there for future projects. Not only that, but having no one to deliberate with and bounce ideas off of was tough since I was unsure if my approach was the most efficient way to tackle the problem. I had to trust my judgement and then if things did not work out (or seemed like dead-ends) I would have to go back and try a different approach. This added extensively to the time it took for me to complete the final project. Whenever I ran into bugs, it took me quite a while to resolve them myself since I had to scan each line of code across multiple files to determine where the issue arose. I also learned the importance of high cohesion and low coupling. When I implemented my Human and Computer subclasses, and was coding all of the methods to my Player class, I realized how much time I had saved by using Inheritance and how planning ahead had benefited me greatly.

(b) If I had the opportunity to start over, I would have been more aware of memory management when I'm creating/deleting/moving objects. After I got the functionality of my program down so that all of the features worked, I ran valgrind and had a very, very, very long list of memory leaks. It took two full days to sort through all of the leaks, locate where they were happening, and ways to delete them without messing up other parts of my code. My biggest problem was that I was passing around card pointers to multiple vectors of Card pointers, so that when I attempted to delete all of the cards I was double freeing. If I deleted a card too early, then I was accessing it somewhere else and was unable to read the data from that memory address. If I had implemented smart pointers or had taken better care of following where I was sending all of my Card objects I could have saved a lot of time that was spent on bug fixing. I was naïve and had assumed that fixing the memory issues at the end wouldn't be that big of a deal, but I will never make that mistake again.

Conclusion

Overall, I was able to take key aspects of what was learned during the semester and apply them to my final project Straights. I learned a lot about designing and coding a large project by working through this assignment step by step and carefully planning ahead.