CS 246 Final Project ~ Straights
Final Project Documentation
Lavan Nithianandarajah
Decemeber 15, 2021

Overview:

In this section I will discuss how my program works on a high level and the structure of the program itself. I will break the program down into individual components to better explain the structure.

Start of Game:
The deck is initialized in main.cc and is shuffled once based on the command line argument provided, if none are provided seed is set to time. We create a Game Object to store the all players, deck, and the four suit piles (Clubs, Diamond, Hearts, Spades). Once the Game Object is created and the invitePlayer() from the Game Class is called to determine the playerType, the Game loop is entered which is the first while loop in main.cc. The dealDeck() function is called to distribute the cards among the players.

Game Loop:
Once the Game loop is entered there are 2 inner loops. The first loop is to loop through a round. The second loop is to loop a player's command in case they input an invalid command. The new round message is printed and the first player to go is also printed. It then immediately enters the Round loop.

Round Loop:
The first function call in the Round loop is the playerTurnPrint() which is part of the Game class. This is responsible for displaying the 4 piles on the table (clubs pile, diamonds pile, hearts pile, spades pile). Additionally, it prints the current player's hand and the legal plays they can play which is calculated from the showLegalPlays() function from the Player Class. Note that showLegalPlays uses legalPlays() as an argument which provides **ALL** plays considered legal if obtained.

The playerType is checked and if the playerType is a Computer Player then the next if condition is entered, otherwise the else statement is entered. This allows the Computer to make automated plays without any inputs from standard input.

Computer Moves:
The Computer Player only has 2 moves to undergo. The first one is play and the second is discard. We first check if the Computer has any legal moves by calling the function legalPlayExist() from the Player Class which returns true if there is and false otherwise. If the condition is true the play card condition will be entered.

Play Command:
Since there is a legal play in the Computer Player's Hand they will now play the first legal Card in their hand and the playFirstLegalCard() function is called from the Computer Class. This function returns the first Legal Card in the Computer Player's hand and removes it from their hand. The returned Card will then be added to its respective suit pile based on the Card's suit.

Discard Command:
If there are no legal plays in the Computer Player's Hand then they must discard the first card in their hand. The discardfirCard() function is called from the Computer Class. In this function the first card is discarded by adding it to the playerDiscard and removing it from the playerHand.

Human Moves:
The Human Player has 5 commands that can be inputted which are quit, deck, play, discard, and ragequit. The player loop will then be immediately entered.

Player Loop: User must enter a valid command to exit the loop.
Quit command: Game is terminated immediately.
Deck command: The current deck will be printed on the screen
Play command:
If the player enters this command then they will also enter in a card. The function playCard() is called. In this function the inputted card will be compared to all the legal plays for the specific player using the legalPlays() function. If there exists a legal play and if the inputted card is found in the list of cards of legal plays, then the inputted card is returned and is added to the appropriate pile based on suit and the Player loop will be exited. Otherwise a Blank card is returned which is a rank of 0 and suit of '0'. If a blankCard is returned the "This is not a legal play" message is printed. The Player loop will not be exited.
Discard Command:
Following this command will be an inputted card. The legalPlaysExits() function will be called and if true that the "You have a legal play. You may not discard." is printed and the Player loop will not be exited. Otherwise the inputted card will be added to the playerDiscard using the discardCard() function in the Human Class and Player loop will be exited.
Rage Quit Command:
The rageQuit() function will be called from the Human Class and a Computer will take the data fields of the Player and print the message "Player<x> reagequits. A computer will now take over." Note the Player loop will be exited however turn count will be reduced by one so that the computer will not be skipped.

Round Over:
The function allHandEmpty() will be called from the Game Class. If true that means all players have no cards to play and the appropriate round end messages will be printed. If that's the case, the above80() function is called from the Game class to determine if anyone has accumulated 80 or more points. If the return value is true the lowestScore is calculated from the lowestScore() function in the Game class and that number is passed into the printWinner() function in the game class to display the winners. If not, the next round begins with the newRound() function in the Game Class. Then the round loop will be exited and return back to the Game Loop.

Design

General Overview of Design of Program:
Throughout the planning phases of this final project it was very clear that this would be a large program handling numerous responsibilities at once. Instantly, I knew that I must modularize these responsibilities and group them together using Object Oriented Programming Principles.

Use of Objects:
Constructing the uml diagram for DD1 of this project clearly outlined the need for Objects. Objects in my design were used as a solution to store similar functions and attributes relating to a specific component of the game. For example, in the game Straights we will need Cards, 52 Cards to be exact. Therefore, I extracted this component to its own Class which will handle methods that relate specifically to the Cards such as identifying the suit of a card or its rank. I extracted each and every component of the game into their respective classes such as the Game class, the Player class, and the Card class. This solved the design problem I was facing which was the lack of modularization. This solution maximises cohesion as each and every object serves a singular purpose working towards the same goal. The methods within these objects are individual responsibilities carried out by the game. This technique minimizes coupling as modules will not need to depend on one another. Change in these functions will not affect other places of use.

Use of Inheritance:
When planning out the implementation of this program it was outlined that we could have 2 possible types of players, one being human, and the other being computer. These two seperate classes shared many similar features, such as methods and attributes. Both Human and Computer have a playerHand, playerDsicard, playerScore, playerType, playerId. They also both have sharing functions, that being, isHandEmpty(), isWinner(), legalPlayExists(), showLegalPlays, and calcScoreFromDiscard(). Without the use of inheritance I would have to copy and paste these similarities in both class. However, using inheritance I created a parent class called Player and made two sub/child classes, one for Human, and another for Computer that both inherit from Player. That way I solved the design problem of Code Duplication.

Use of Polymorphism:
Constructing the Game Object I needed to keep track of all the players in a vector that are part of the game whether they be Human or Computer. However, I can’t store all the players as Human as there are Computer players and likewise I can’t store all the players as Computer because we can also have Human players. This is when I used Polymorphism to counter this design problem. Human and Computer are polymorphic as they can take the type of Player. Therefore, I stored the Human and Computer players in a vector of type Player. Due to Polymorphism this implementation is completely legal and allowed me to keep track of all the players in the game regardless if they were Human or Computer.

Resilience To Change

General Overview Resilience to Change:
Due to the modular structure to my final straights project, each responsibility is divided into its individual functions. For example, finding the first player is determined by a single function called firstPlayerId() in the game class. Likewise, there are functions in their respective classes in order to divide individual responsibilities of the straights program itself. If there were any changes to the game logic or program specification the entire program will not need to be re-implemented, just the functions that are responsible for those specific specification changes. Once these individual methods are changed the straights program itself will be able to work.

Possible Changes to the Project Specification and How my Program is Resilient
Although there can be hundreds of changes to the project specifications, I will analyze my program's resilience to the most common changes to game logic utilizing cohesion and eliminating coupling.

Change in Legal Plays:
A common change to games is the rules of the game itself and what is allied to be played in order to progress through the game. If there are changes to what is considered a legal play in the straights project the only change I will need to make for my program to work with the changes to game logic is in the legalPlays() method in my Game class. This method alone is responsible for determining what exactly a legal play is. For example, if the legal play changed so that we can play an A if there is a K played or if we can play a K if there has been A played. This will be determined in my legalPlays() method and the return value will contain all the possible legal plays. These legal plays will then be used for the rest of the program and the program will work for the changes in specifications.

Change to Who the First Player is:
Before any round starts, we must determine who the first player is. As of now the first player is the one that possesses the 7S (seven of spades). However, let's say that the first player now will be determined by who possesses the 5C. The only change we will need to make is in the firstPlayerId() method found in the Game class.

Change in Addition of Jokers:
The current specifications exclude the Jokers from the deck of cards. If we were to implement the addition of Jokers to the deck of cards. We will need to only make a few changes. The fundamental change that must occur is in our Card Class as we must now be able to Output and Input Joker cards. The next change will be in the initialization of the deck as now we must include the Jokers as well as the dealing of Cards. The remaining change will be in terms of what the joker allows you to do. For example, if the Joker is a wild card, then this must be handled in the legalPlays() method in Game class. But with only a few changes our entire program will compile and run successfully.

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.

Answer: Changing the interface from text base to test-base to graphical is a massive change in the visual representation and will cause re-implementation of the majority of the **Game class**. To have as little impact on the code as possible I would implement the **Strategy Design Pattern**. This will allow us to display our game in two separate ways depending on what the client desires. In this case, the algorithms are used for displaying purposes and are encapsulated which makes them interchangeable. This benefits us as the programmer since these displaying algorithms can vary independently without having the need to change any other code outside the **GraphicsDisplay Class** and the **TextDisplay Class**. Before this change, our **Game Class** was responsible to implement display whether it be graphical or text-base. Any changes to the methods used for displaying will cause the entire **Game Class** to change. Having this **Strategy Design Pattern** implemented the **Game** and the **Display Class** will live independently therefore changing from text-base display to Graphical display will not affect game logic significantly. Note that the **Display Class** will be the parent of the **GraphicsDisplay Class** and the **TextDisplay Class.**

In terms of changing the rules of the game the impact of the changes will cause a great impact on my planning state of the game. However, my program is now resilient to many changes in Game logic which will briefly be outlined in the final paragraph of this question and thoroughly explained in the "Resilience to Change" component of my design.pdf document.

In order to minimize the amount of code I will have to change from upcoming rule changes, I would change my class design so that I extract my entire game logic involved with rules into its own class and call this class's methods when I need them during the game rounds. That way the rules of the game live independent from the Players, Deck, and the Card piles too. The changes to the rules of the game will only affect the **GameRules Class** and we will not need to re-implement them in any other calls besides those which means it reduces the amount of code we will need to change.

My final straights project does a great job of extracting the legal plays based on the cards that have been played on the table. This is because all legal plays are calculated in a single function called legalPlays() found in the **Game class**. Any changes to the rules regarding determining the legal plays will only affect the legalPlays() method in the **Game class**. Once legalPlays() is re-implemented to fit the current rules regarding legal moves the program itself will work. Likewise, for determining the first player, that logic is handled in the firstPlayerId() method in Game Class. Any changes to how a firstPlayer is determined

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?

Answer: Currently under the **Computer class** the only methods that control the game logic are the **playFirsLegalCard()** and **discardFirstCard().** These methods are quite simple as we just need to find the first legal play in the computer hand of cards or we must find the first card in their hand. This algorithm can be changed to introduce more strategy into the game. For dynamically changing computer strategy we can add more methods based on the STAGE of the game. The STAGE of the game can be determined by the number of cards that have been played on the table and this attribute can be a datafield in the **Game class**. We can define the first third as if there have been 1-17 cards played. The middle of the game can be defined as 18-35 cards played, and late game being from 36-52 cards being played. Based on these 3 stages of the game the Computer Strategy can alter. Each of these stages will have their own **discardCard()** and **playCard()** methods that take the form of **lateGameDiscard()** or **lateGamePlay()**. Within these methods is where the alternate computer strategy can be implemented.

An idea to make the Computer Player logic more sophisticated can be the fact that instead of just discarding the first legal card in their hand, we can check the STAGE of the game and if this stage is late game then the computer will discard the lowest ranked card in their hand. If the game is still in the Early STAGE then it will discard as normal which is the first card in their hand.

Likewise, we can take a similar approach to the playing of a card by a Computer Player. Instead of simply playing the first legal play in their hand, we can check the STAGE of the game and if it's a late game we can play the highest ranked legal play card. This will be a more sophisticated strategy as that means there are less higher ranked cards remaining in the Computer player's hand as a potential discard further minimizing their overall score giving them a higher probability to win. These methods will allow the computer player to change strategy as the game progresses.

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?

Answer: Although this is a major change to the rules of the game the flow of it is more or less the same with the options opened up to the 2 Jokers. The major changes to the code will occur in the legal plays section of the program as that is what controls what cards can be played on the existing pile. With the addition of jokers we must now also consider that as a legal play on top of any card. Since the Joker is represented as a wild card, meaning it can be any card besides the 7 of spades, we must confirm the rank and suit of the Card that the Joker converts to to continue the flow of the game as the Joker itself does not have a playable rank or suit. Therefore we will need to add a control flow statement (if condition) saying if the played card is Joker then list out all legal plays aside from Joker and number them from 1… and the player must input a 1 that corresponds to the legal play and the Joker card will be "destroyed" and card the player would like to convert their Joker to will be created. This process will occur under the **Human class** and the method **playCard()**. Additionally, we must also include Jokers as a legal play under the **Human class** under the **showLegalPlays().**

Another factor of the design that will slightly ulter is the deck initialization and the deck capacity. Currently the **deck** contains 52 cards however with Jokers it will become 54 cards. Another factor to consider is the classification of the Joker in terms of Rank and Suit found in the **Card class**. This can be represented as 14 in rank and J in suit or any other approach as long as the characters is not used by any other rank or suit. Aside from that the design will be kept the same and would be an amazing feature to try out as a bonus.

Final Questions

Lessons I Learned:

I have learned a lot from completing this final project as in encorporated all aspects of this course into one final program. My main takeaway from the final project is that whenever designing large programs, design is absolutely crucial. It is almost required to think critically before actually typing even one line of code. Additionally, I was not only able to understand principles of modularization and object-oriented programming but was able to apply this knowledge at a large scale to create a game program.

Importance of Understanding Structure before Programming:

Although, it was tedious to come up with a plan of attack and a uml diagram of our program before actually coding, it definitely was a helping hand in guiding students in the right direction for completing this final project. There were countless times where I referred back to my uml diagram in order to understand relationships between classes and the overall structure of my program. Additionally, the plan of attack itself kept me on track to complete my project as it had specific personal due dates and all the components I needed to complete. Without outlining the program structure and how I would complete this project, I would have wasted a lot of time just trying to understand how to structure my program with disconnected pieces of code.

Importance of Modularization and Object-Oriented Programming:

When it comes to large programs it is absolutely crucial to divide the responsibilities of the program as much as possible. Grouping these responsibilities into Classes and within these classes will be the methods that undergo the program's responsibilities. Before taking this course, modularization was simply done by dividing the program into functions and managing these return values in main as variables. This leads to main being filled with too many variables to efficiently manage. Furthermore, functions that depend on each will often have repeated code, in other word code duplication. If you change one function you will need to change every single place you use that function which is incredibly time-consuming for large-scale. Applying Object-Oriented programming to this final project allowed me to group similar functions and variables together as objects(maximizing cohesion), removing code duplication and limiting interdependencies between functions(minimizing coupling). Any changes within these objects will not affect the interface of the program that interacts with these objects which will save a lot resolving changes. The application of Inheritance taught me how to reduce code duplication even further. For example, the Computer Class and Human Class share all the same attributes and some methods. Creating a Player Class as a parent class for the Human and Computer class removed having to write shared traits between these classes. The application of modularization and Object-Oriented programming made my implementation of the code concise and simpler to manage, saving many hours if not days to complete.

What Would I Do Differently Next Time:

Although I was able to complete the final project on-time there would be some aspects of this project I would do differently if I had the chance to start over. First and foremost, I would have tested each of my Classes independently and made sure they worked perfectly on their own before making Classes work with one another. Another aspect of my final submission I would change is design my test harness (my main) to contain as little code as possible. Another change I would make for next time would be to design a Round Class.

Why Test Classes Individually?

When implementing my final project I created individual classes and once I was done implementing them I moved onto the next class as that is what I outlined in my plan of attack. However, I did not know if these Classes contained any logical errors or not. The only information I knew was that there were no compilation errors. However, when I completed all my individual classes I started implementing the tes harness which combines all these classes together and actually creates the Straights game itself. The test harness was built of the building blocks of the Classes and with wrong implementation of these classes the game would simply not work as anticipated. As a result of this, I spent a lot of time trying to figure out where the error may have occurred and how to possibly fix it. This wasted time could have been avoided if I simply created a test harness to test each Class on its own before moving onto the next Class. Although I was able to find these logical errors and fix them with correct implementation, I wasted a lot of time that could have been used to implement bonus features I had in mind.

Why Keep Lines of Code In Main to a Minimal?

Even though I applied object-oriented design my main.cc was relatively large. Keeping main.cc concise simply promotes code readability. Most of these actions such as creating attributes of the Game Class to instantiate a Game Object could have been done in the Game Object Constructor. Although the program still works, keeping the main.cc clean allows me to revisit my project in the future and understand the flow of my program much faster compared to if main.cc was long.

Why Create a Round Class?

Analyzing my Game Class implementation it was clear that my Game class was doing a significant share of the responsibility of how the straights program works. This was expected however, the Game Class's responsibility can be divided further into another Class called the Round Class. Breaking down the Straights game we know that a round ends when all players have played or discarded their Cards. At the end of a round, player scores are calculated and if anyone exceeds 80 points the winners are determined and the game is over. If not a new round starts, where the table is reset and the deck is shuffled and redistributed. If I create a Round Class, I can create methods to shuffle and distribute the deck, add player scores, and determine if there's a winner. Additionally, I will be able to keep track of the 4 suit piles (Clubs, Diamonds, Hearts, Spades) independently. This will transfer some of the responsibilities of the Game class which is currently managing all of these methods and attributes.