Design

Please note the use of class methods without their parameters for readability

1 Overview

The Card class is used to create a card using two Enum classes, Suit, and Rank which represents one of four suits and one of thirteen ranks respectively. The Deck class has an instance of a vector of cards. This is initialized to have 52 cards when constructed that represent a real-life deck of cards. The Deck class has a shuffle method and a method to get a card based on the index at which it is stored.

The Table class has 4 instances of a vector of Cards where each one represents one suit. The Table class stores and updates the four piles of cards based on the suit and rank as the game progresses. It also has methods to check if a card is legal or not, add a card to the table and calculate legal plays when given a vector of cards that represents a player's hand. It essentially controls if and where a card should be placed in one of the four piles.

The Player class is an abstract class from which the Human and Computer class inherit from. The Player class keeps score and stores and updates a player's hand and discards as the game progresses. There is dependency between the Player and Table classes because the Player Class takes in the Table Class as a parameter in its pure virtual methods. These two methods also take in a reference to a card. This is done so that the Human and Computer concrete classes can play and discard a card on their own using the Table class. Note that the game calls the play and discard method using the same method for both classes.

The Game class has an instance of the Deck, Table, and 4 Player classes and uses these three classes to implement its own methods. It controls when a player should move to the next player, if a human or computer player is created, when a round ends, whether the target score has been reached, how to reset a round, when to shuffle the cards and other logic associated with the game. Moreover, it has getter methods implemented using the above three classes and its own implementation. For example, to get the deck, the table, the hand, the discards, and legal plays, it uses the Deck, Player, and Table classes. And to get the winners, the turn of a player and the scores, it uses its own implementation. It is essentially the Game class that ties in the other three classes and controls the logic of the game. It also has the target score, number of players, current player, first player and seed as fields since these variables or constants are part of the straights game.

The Message Class consists of all the output messages specified in the starights.pdf project. It is dependent upon the Card and Table classes since some of its methods take the Card or Table class as parameters. The User class controls some the flow of the game, all the input syntax and output messages to be printed. I say some of the flow because the User Class does not have complete control. For example, the User class cannot move to the next player. That only happens if a Player has successfully discarded or played. This is controlled by the Game Class and not the User Class. This design choice was intended to practice encapsulation and restrict the User from making changes that would not comply with the internal implementation of the Game class. Just like the Table uses the Cards methods, the User can use the Game's methods to play the game.

The following is a brief description on how the Game classes' play method works. Note that the User class knows if the Player is Human or Computer. The reason for this is explained in the design section of the document. Assume that a card is passed to the Game play method. The Game class then passes in the Card and Table field as references to the play method of the Player Class. If it is a Human class, the play method will check is the play is legal using the checkLegal method in the Table class. If not, it will return false. If it is legal, the Human class will add the card to the table, remove the card from its hand and return true. If it is a Computer player, the class will use the findMove method to find the first legal play based on the hand. If no legal play is possible, it will return false. Otherwise, it will change the card reference to the legal play and return true. The bool returned from the Player's play method will be the same bool returned by the Game's play method.

2 Design

Challenges

For type safety and to avoid the implicit conversion of an Enum to an integer I make use of Enum classes. As seen in my UML I have many getters. To prevent the change of internal fields and not repeatedly copy to the stack, I used a return type of const reference for these getters. To not have pointers throughout my program, but still allocate memory in the Heap, I allocate memory for a new Game in the User class only. Thereby, all subsequent classes in the Game class will be allocated in the heap. To prevent object slicing but still have a single Player class array, I create a vector array that stores pointers to the Player class.

My program could have worked without the Table Class if I decided to include the four piles in the Game Class. The reason I thought the Game Class should have the piles is because the Table can be thought of as a part of the Game. That is, even though, it is possible for the table to exist without Game, it would not have any real function outside the game of straights. However, this is also true for the Player class. Only a player of the straights game has a discard pile, a play and discard method, and a specific way of keeping score. I realized that in such a program, it would make sense and benefit the program to have a Table Class. I also realized that the if I were to fit all the components of the table in the Game class, the Game class would be doing too much and there would be no delegation. This would also affect the size of the Game Class for the worse. And in the end the table class is independent from the Game class, even if it does not make sense for it to exist on its own. For these reasons, I decided to create the class.

The Human and Computer classes inherit from the Player Class and have their own play and discard methods. To practice perfect inheritance, the player and computer class would need to play or discard their turn without the User Class or Game Class knowing the type. But since a human player decides to whether to play, discard or output a message depending upon if a move is legal or not, I would need to inlcude input, std::cin and print output, std::cout from within the Human class. This would make it tougher to change the input syntax and output messages. Moreover, the Human class would need access to the Game class to be able to output the deck and table or ragequit. While this is perfectly fine choice, I wanted to separate the input from the output. Having this separation makes it easier to read and the Player class does not have access to the Game class. To solve this, I did two things. First, I introduced a variable called type in the Player Class, that would be true for a Human and false for a Computer. Second, I passed in the reference to the card and the Table in both the play and discard methods for the Player class. This way both classes can make the necessary changes to itself, the card and the table still practicing inheritance and not having access to whole game class. The Game class calls the same discard and play method for both of Player's subclasses, it is the User who needs to know whether the payer is Human to output a prompt or expect a command. Although there is dependency between the Player and Table Class, this is necessary so that a Player class may play their move without the interference of the Game class.

Another challenge I faced was the decision on the amount of control the User class should have on the game. I wanted the User to have some control but at the same time, I wanted to make it such that there is a smaller chance for error. Therefore, most methods in the Game class do not take in a parameter and correspond to the variable curr. This variable represents the current player. This way a User cannot play, discard, output the hand, and more for Player4 when it is Player2's turn to play. I also have a method to get the turn of a player so that a user is aware whose turn it is and can print the correct message when needed.

As mentioned previously, I wanted to separate the input, output, and logic of the program. To do so, I created a Message class that stored all the output and that has no relation to the Game. It would merely output a Table, Card, or pile in a predetermined way. Moreover, I created the User class and structured the Game class such

that the User would only need to use the public methods to play the Game. This also allowed for the User to define their own input syntax. Therefore, the User has control over the input and output.

Another challenge I faced was keeping track of the scores of players for the previous and current rounds. To do so, the Player class keeps track of the current round scores, while the Game class is responsible for storing the previous scores in a vector int array. After every round, the resetRound method is used to update the score array in the Game class using the Player class's getScore method for that round.

UML revisions

In my final UML design, there are four major things to note. First, the name of the View class is changed to User class. The initial name for this class did not indicate the intent of its use. However, the idea behind the User class is the same. It controls the execution, input, and output of the game. Moreover, there is no enum class for the commands. I found this unnecessary and easier to understand and implement without the enum class. Second, there is a Message class. Initially, I had all the output messages in the User class but realized it would be better if all the output was in one class. This increases cohesion and decreases coupling. I do not need to recompile the entire User class if I decide to change an output message.

Third, the getLegalPlays method is shifted from the Game class to the Table class. I did this because as mentioned previously, the Table is like an extension of the Game. However, the main reason is because my addToTable method in the Table first checks if a play is legal before adding it. Moreover, the Table class has all four piles of cards making it easier to implement the getLegalPlays method in the Table class. And so, I was able to utilize this to find the legal plays for a player. Lastly, there is no dependency created between the player and Table class since the method takes in a vector of Cards. The Game class decides which players legal plays should be calculated. Another change to note is the removal of id from the Player class. This is because I can simply use the vector array index or current player variable of the Game class to calculate the turn of a player.

Fourth, the relationships between the classes are still the same, only shown in a different way in the UML. That is, the Game still has an instance of a Deck, a Table, and 4 Players. The Human and Player class still inherit from the Player abstract class. The Player, Table and Deck still use the Card class. What did change, is the dependency relationships and numbers or fields and methods for each class. I could have not foreseen the dependency between the Message class and the Table and Card since I did not plan on implementing the Message class at the time. I did also not anticipate the dependency relationship between the Player and Table class to be able to implement inheritance correctly. Lastly, I honestly underestimated the complexity of the program which explains the lack of methods in the initial UML.

3 Resilience to Change

Input syntax

Assume that you want to change the input syntax for the Card Class from <rank><suit> to <suit><rank>. Because I used a >> operator method in the Card class, I would only need to modify this. The rank and suit would still be stored in the same way and since all classes do not have access to the internal fields but the public methods, only the card class will be recompiled. Assume that you want to change the commands syntax to their first letter. That is play will become p, discard will become d and so on. This is a trivial change and can be done in the User class since all five commands are present in this class.

Output messages

Assume that you want to change the order of suits printed in the table class. To do so, there would be no change in the Table class, rather the Message class method to print the table would only need to be changed. This is because the User gets the Table from the Game class and passes it to the Message class method to print. Assume that you want to simply change any given prompt in the project. Again, the only change that would take place would be in the Message class since all the output prompts and messages are stored in this class. Assume that you want to change the output syntax of card from <rank><suit> to <suit><rank>. As before, the << operator method would only need to be updated for the same reason as the >> operator.

Rules

Assume that the legal plays specification for a player changes. To accommodate this change, I would simply change the checkLegal method is the Table class. Since the method to add a card to the Table class and the method to find legal plays are both present in the Table class and use the checkLegal method, this change will be reflected throughout the program. Assume that you want to change the target score for the game. This can simply be done by reflecting this change in the Game class constructor. Assume that the start card is not the seven of spades. To make a card of your choice to be the start card, you would simply need to change the const start cards' value in the Table class. I placed the start card here because my Table class checks if a play is legal or not, thereby having high cohesion in the class. And since, the Game class includes the header file for the Table class it will have access to the start card without the need for recompilation. Assume that you want the game to end after two rounds whether the target score is met or not. To do so, you could either make changes to the current User class or create a new User class where the program ends when the second round is reached. This can be done because there is a method to see if a round has ended or not in the Game class. Assume that you wanted to shuffle the deck once the first round, twice the second round and so on. This can be simply done in the Game class. I would separate the shuffleAndDeal method to shuffle and deal methods so that the user can shuffle the cards many times they would like. I did not do this in the program to limit the number of times a User can shuffle the cards. Currently, the User class would implement this by calling the resetRound and shuffleAndDeal methods. The resetRound clears the players hand, discards and scores. Therefore, if not called each player would have more than 13 cards.

Players

Assume that you want 2 to 4 players to be able to play. To accommodate this change, I would first need to create a constructor for the Game class that takes in the number of players and not just the seed. By doing only this, the change will be reflected for 2 players since it is an even number like 4 and a round ends when all the players have zero cards, that is the Player class's hand fields are empty. However, to account for 3 players, I would need to change the condition for which a round ends in public roundNewOrOver method. I would simply add the condition that if there are three players and only one has a card left then the round ends. Another possible solution would be to play the starting card automatically when dealing the cards to the players and the player who previously had the seven of spades would start first. This way there would be no changes in the roundNewOrOver method but the shuffleAndDeal method would need to be updated since this method assigns the first and current player at the start of a game and adds cards to the players hands. While dealing the cards, I would add the seven of Spades to the table and assign the current and first payer variable to the payer with the card. Lastly, the User class would need to be updated for outputting the right number of prompts in the beginning of the game.

Versions

Assume that you want multiple versions of the straights game where the input syntax, output messages and flow and number of players can differ. This can simply be implemented by creating different User classes where one can decide the input syntax, what should be outputted and when using the Message Class, and how many players should play the game when initializing the Game as mentioned above.

Features

Assume that you want the computer player to not just play the first legal move but a calculated advanced play or discard move. The play move will still be rather simple, where the Computer just plays all cards of rank seven first. If no ranks of seven cards are in their hand, it will play the first legal play. However, when the computer must discard, it will do so with the card that has the least points. To accommodate this change, I would need to do the following. Currently, I have a private helper method findMove for the play and discard methods. This method attempts to find the first legal play possible. If true it will return true and change the card reference parameter to the legal play else, it will return false. Then the Computer will call the discard and the first card in the hand will be played. I would update the findMove method to first find any card with the rank seven using the getRank method of the Card class. If no such rank is available and the legal plays are possible, the first card will be used to play the move. However, if the findMove method returns false because there are no legal plays, the discard method will play the card with least value. I can implement this using the getValue method of the Card class. Finally, this new feature will have been implemented and the only change would be in the computer class demonstrating low coupling and high cohesion. Assume that you want a progress bar tells players how many cards were played or discarded out of the deck. To implement this, I would need to create a method in the Game class only. This method would calculate a number out of 52 by adding all the cards in the Table class using the getTableSize method and all the discard piles for each player using the getDiscards method. I could then use the NCurses library to implement a progress bar based on the number returned and the total cards.

Interface

Note that I did not solve this program with the intention to implement a graphical interface. However, assume that you want a graphical interface for the program. What I imagine I would have to do is create two classes one for the text and one for graphics. The Text class would be similar to the Message class but have a notify method with a state parameter to call the right output. The Graphics class would also have a notify method but instead display graphics based on the state of the game. I cannot be sure of the implementation since I didn't plan on implementing a graphical interface. But I do believe that most of my classes will stay the same.

4 Coupling, Cohesion and Recompilation

When implementing my program, I spent a long time planning out the classes and their interactions to minimize coupling and maximize cohesion. I also kept in my mind that minimal recompilation will follow if I am able to practice low coupling and high cohesion in my design. As demonstrated in the design and resilience to change sections, I make minimal modification to change major and minor aspects of the program.

Furthermore, we see that when accommodating change, only one class is changed and recompiled provided the change is relevant to that class. Also as seen in the UML, methods relevant to a class are implemented in that class. This is because I wanted each class to do one task well following the single responsibility principle. For example, the Deck Class has the shuffle method and not the Game class and the Player has relevant methods like play, discard, and addToDiscards. For these reasons, the program practices high cohesion.

Also notice, that all classes are directly or indirectly dependent on the Card class. But due to the specification and nature of the program, this is necessary. However, excluding the Card class, we see that the Table, Message, Deck, and Player Classes are all independent. The dependencies that do exist between the Player and Table class and Message and Card, Table classes are weak dependencies. The Game class has a Deck, Game and 4 Players while the User class has a Game and a Message class. These relationships are necessary for the program to work even if they cause dependencies. For these reasons, the program practices low coupling.

5 Answers to Questions

Question 1

The MVC pattern should be used to structure the game classes. This allows additional concrete View classes to be added such there is a text-view class and graphical-view class. In the MVC the logic, output, and input are separated using the Model, View and Controller classes respectively. The controller would read from std::cin and the view would store and print to std::cout. The text and graphical View classes are notified when the Model, that is the Game class's states changes. When changing the rules in the Game class, the Controller and View would be easily updated to reflect this change. My classes do not fit this framework since I did not intend to add a graphical interface. However, as demonstrated in the resilience to change section, there is little impact to the code when changing the game rules.

Question 2

I would make the Computer class abstract so that different type of Computer players may be initialized. As explained earlier, in the resilience to change section, my code would require changes only in the Computer class to play more advanced moves. It is similar in this case as well. However, the method to find a move to play or discard will be implemented in the Computer subclasses and not in the Parent class. And because the Computer play and discard method have access to the Game Table, its' subclasses will also have access to it and can use it to differ play strategies and dynamically change them as the game progresses. Therefore, only the Computer class would need to be modified and its subclasses created to accommodate for this change.

Question 3

The design of the program would not change. However, there would be changes to some classes. I would need to add the Joker as a valid card in the Card class. Moreover, the Deck class will now consist of 54 cards instead of 52. The Table class does not need to change since, the Joker takes the place of one of the 52 cards. However, the Game class would need to be able to handle the Joker as a wild card. That is, it would have a method that takes in the Joker card and the desired card to play or discard. Lastly, the User class would need to reflect this change by having a new prompt to play or discard a wildcard. For example, <wildcard> <JK> <TS>. Lastly, the Game class would need to update the roundNewOrOver method to when two players have one card each. If the output must be in the form that the <JK> card gets printed and not the wildcard intended <TS>, then the addToTable method in the Table class would need to be updated.

6 Final Questions

Question (a)

I learnt that organization, time-management, and planning was crucial to successfully implement a large program. I also learnt the importance of delegating work between modules while still maintaining low coupling and high cohesion. I understand now that there is no such thing as perfect design, and one must give up on one aspect for another depending upon what is needed. One of the most important things I learnt, is to plan and break up the work. Moreover, I learnt to keep testing code as you write, whether it be a method or class. This helped me a lot. Moreover, using a debugging tool such as gdb is essential when writing large programs.

Question (b)

If I could start over again, I would do two things. Whenever I made a design choice, I would write down what problem I faced and why I chose to make this choice. This would help me keep track of my thought process. And I would not spend as much as time on the theoretical side of the design and implementation. This is because I found that as I wrote more code, I became aware of more of the design choices needed to be made, the implementation needed to solve the project and why an earlier approach was not going to work.