1 Introduction

I chose to implement the cards game, Straights for the CS246 project requirement. Please note that I do not follow a class's name with the keyword class every time I refer to it (e.g. Game, Table). Additionally, methods are italicized and parameters are excluded to better comprehensibility when reading (e.g. *makeMove()*, *addToTable()*).

2 Overview

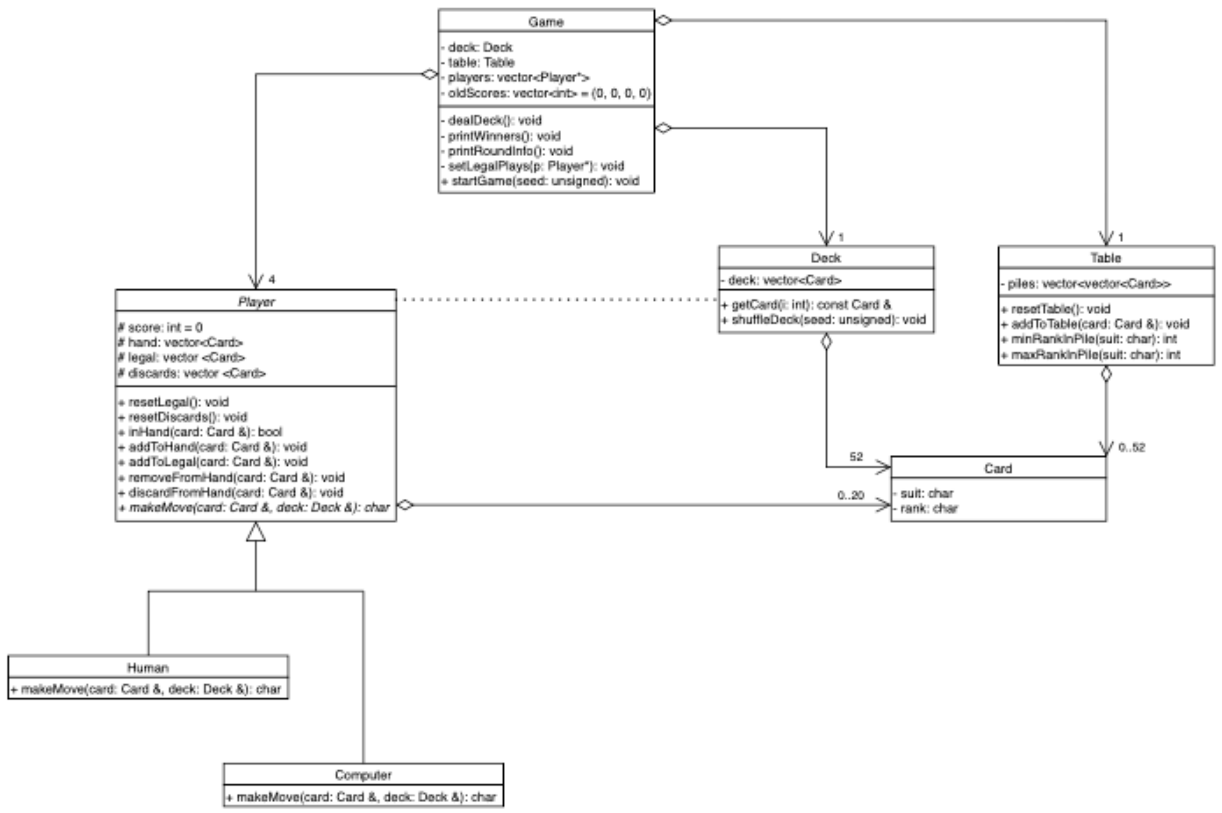
Game
- deck: Deck
- table: Table
- players: vector<Player*>
- oldScores: vector<int> = (0, 0, 0, 0)
- dealDeck(): void
- printWinners(): void
- printRoundInfo(): void
- setLegalPlays(p: Player*): void
+ startGame(seed: unsigned): void
1
1
4
Deck
- deck: vector<Card>
+ getCard(i: int): const Card &
+ shuffleDeck(seed: unsigned): void
Table
- piles: vector<vector<Card>>
+ resetTable(): void
+ addToTable(card: Card &): void
+ minRankInPile(suit: char): int
+ maxRankInPile(suit: char): int
Player
# score: int = 0
# hand: vector<Card>
# legal: vector <Card>
# discards: vector <Card>
+ resetLegal(): void
+ resetDiscards(): void
+ inHand(card: Card &): bool
+ addToHand(card: Card &): void
+ addToLegal(card: Card &): void
+ removeFromHand(card: Card &): void
+ discardFromHand(card: Card &): void
+ makeMove(card: Card &, deck: Deck &): char
52
0..52
0..20
Card
- suit: char
- rank: char
Human
+ makeMove(card: Card &, deck: Deck &): char
Computer
+ makeMove(card: Card &, deck: Deck &): char

Figure 1: Final UML design

The UML figure above demonstrates my final design efforts. Note that Vector STL's are used in Deck, Table, Player and Game and my implementation does not make use of any design pattern. In contrast with my initial UML design, this one consists of a Table instead of a Pile. Another noteworthy change is the replacement of the virtual methods *doPlay()* and *doDiscard()* with a single virtual method *makeMove()*. Additionally, the multiplicity between classes have been updated, default values for *score* in Player and *oldScores()* in Game have been noted, and a new dependency between Player and Deck has been highlighted. Finally, the vast number of member methods that served to print information have been replaced by way of operator overloading. For example, instead of a *printDeck()* method like in my previous UML design, I overload the operator << for Deck.

The order in which I will present an overview of my major classes is as follows: Card, Deck, Table, Player, Computer, Human and finally Game. This is in an order such that the modules corresponding to the classes with the least compilation dependencies are described first.

Card is not dependent on any class, but all the other classes are dependent on Card. Deck, Table and Player have direct dependencies to Card. Deck has a vector of 52 cards. Table has a vector of vector of cards amounting to 0 to 52 cards at any given point in the game. Player has three separate vectors of cards all of which total to 0 to 20 cards. All three classes have an aggregation relationship with Card. Note that the multiplicity these classes have is based off the rules of the game. For example, a player may have 12 cards in his hand and a maximum of 8 legal cards for a total of 20 cards. Human and Computer inherit from the Abstract class Player and so have a specialization relationship with Player. Game has 1 Deck, 1 Table and a vector of 4 pointers to Player objects. As a result, it has an aggregate relationship with Deck, Table and Player.

All classes have private or protected member fields and member methods are only made public when necessary. For example, of all the member methods in Game, only *startGame()* is public. Finer details regarding the reasons behind the implementation of specific relationships, and OOP design principles such as encapsulations are noted below.

3 Design

Vector STL's are implemented in Deck, Table, Player and Game to help manage memory implicitly and significantly reduce otherwise redundant code.

Encapsulation in implemented wherever possible to protect the objects from unwanted access or manipulation. For example, all the member methods of Game except *startGame()* are private since only Game needs access to them for the public method *startGame()*. The member fields of Player are protected to prevent accidental or voluntary change to the object but allow Human and Computer to inherit those member fields. Player has been implemented as an abstract class to allow for such inheritance. The method *makeMove()* is a pure virtual method. This allows me to create separate implementations for Human and Computer and call *makeMove()* on a Player object without having to worry of the specific type of class.

In Game, I make use of a vector of pointers to Player objects instead of Players to prevent the use of a vector of polymorphic objects and instead make use of a polymorphic vector of pointers. This prevents any potential misalignment and object slicing.

I make use of a vector of vectors in Table to store the cards. The vector holding the cards of a particular suit is in the order of "CDHS". In other words, the first vector holds cards corresponding to Clubs, the second to Diamonds and so on. This is done to minimize code redundancy and allows for one private member method instead of four. Repeated code is minimized among the Table member methods as well.

Instead of implementing a traditional getter for Deck, *getCard()* returns a constant reference to a particular card for read only access. This excludes the need to copy the entire private member field deck every time Game needs access. This helps save significant memory and improves efficiency.

Setters have been designed to specific in action as well. For example, *addToTable()* does not return the Table by reference for undocumented change (this would not be possible regardless, since the member method, piles is private) but instead adds the card passed as a parameter specifically to a particular pile. This is one such example of a solution to get around encapsulation. Other setters include *addToLegal()* and more.

A similar example are the public member methods *minRankInPi*le() and *maxRankInPile()* in Table. Instead of using a getter to return the Table by value or by a constant reference and then use it to check whether a play is legal, these member methods are designed to return the minimum and maximum rank in a particular pile as the name suggests. What this does is it saves memory, betters efficiency by returning an integer allows and most importantly allows Game to quickly determine whether a card is a legal play.

The method *setLegalPlays()* has been implemented to set the member field legal in Player at every turn in the game. In this method, the legal cards are added to legal, and the method returns void. It could have been the case where the method returns a vector of cards which would then get passed around by means of a parameter. I did not choose to do so as it would increase the use of memory and decrease program efficiency as it got copied from one place to the next.

Note that methods like *getCard(), minRankInPile()* and *maxRankInPile()* increases cohesion significantly as the classes Card is responsible for finding a card at a particular index and the Table figures out the minimum and maximum rank in a particular suit instead of passing on the responsibilities to Game. This makes greater sense as the Game should not be responsible for such implementations, thereby increasing cohesion. Another such example is *discardsFromHand()* in the class Player. I could have implemented separate methods namely, *addToDiscards()* and *addToScore()*. Then, whenever a discard took place, Game would have to call three functions, namely *removeFromHand()*, *addToDiscards()* and *addToScore()*. However, the implementation of *discardFromHand()* does the work of all three functions combined. This once again results in increased cohesion.

Moving on to dependencies and coupling, the chief reason for the dependency between Player and Deck is to allow Human to print the deck class. So, to minimize the dependency, I pass the Game's Deck object as a constant reference to the method *makeMove()*. Moreover, I do not include Deck's header file in Player's or Computer's implementation file. It is only included in Human's implementation. This is possible as I use a forward declaration in Player's header file. The beforementioned details minimize coupling between the classes Player and Deck as well as Computer and Deck.

On another note, I allow the Human class to manage input because of which the dependency to Deck gets created. This was intentional and necessary as there would be no other way to call *makeMove()* without having to know the type of the Player object i.e. whether it is a Human or Computer. To me, the slight increase in dependencies, that too a weak one was worth it as it allowed for the implementation of true inheritance i.e., the type of Player would not need to be tracked.

Finally, all my classes demonstrate minimal coupling. For example, my Card class is independent of any class as it should be. Additionally, Table, Deck and Player are only dependent on only one class i.e., Card. This is necessary due to the nature of the program specifications. Apart from Card, notice that Table, Deck and Player are all independent. On another note, the Game class is dependent on all the classes. However, this is once again necessary as it controls the logic of the game and thus needs access to the classes that make up the game.

Nevertheless, when making changes to the program, dependencies are minimal and individual classes need to be recompiled when changes may be made. For example, a change in *shuffleDeck()* in the class Deck will only require the recompilation of Deck. The same goes with Card, Table, Player, Human, Computer and Game. Consequently, dependencies decrease, and minimal recompilations are required to introduce major or otherwise minor changes to the preexisting program.

Conclusively, all the above mentioned demonstrates a solid object-oriented design, and the practice of minimizing coupling and maximizing cohesion.

4 Resilience to Change

Any program resilient to change must include strategic constants such that major changes can be dealt with relative ease. For example, the game ends when a player has reached a maximum of 25 points. In my program, in Game's implementation file, the constant maxScore is all that needs changing.

Another such example is a change of rules as to which cards are legal. Suppose the game specification changes and the first card to be played is the 5 of Hearts. Not only that, suppose the next legal cards are the 5 of Diamonds, 5 of Spades, 5 of Clubs, 6 of Hearts and 7 of Hearts. The rest of the rules remain the same. Then, the only changes required of my program is to change the constants firstRank and firstSuit to 5 and H respectively in Game's implementation file. This will accommodate the new change of rules. Another change of rules may wish to declare those with the greatest scores the winners. This can also be easily accommodated with a single change of a sign in the Game's private member method *printWinners()*. Say the value of ranks changed, and value of A was 13 and K was 1 and so on, the only constant we would need to change is ranks in the Card class. It would have to be written it revere. Another situation could be presented whereby game ends after a set number of rounds. This would call for minor changes in game's public function *startGame()*. A counter would have to added and the code for updating the variable roundOver in *startGame()* would appropriate minor change.

The last example of a change in rule's would be to accommodate a 2, 3 or 4 players. The changes to be made would be relatively simple, that is, the constant maxPlayers in Game may be changed or otherwise set via a command line argument. That would require a minor change in main's module and a few changes in Game's implementation file. More specifically, it would have to accommodate how many cards are declared to each player which would be a minor change in Game’s member method *dealDeck()*. Additional changes would have to be made to either the Human and Computer classes or the Game class to skip a player’s turn when their hand is empty but other players’ hands are not. The skipping of turns would only be required when the number cards dealt to each player are not the same, i.e., when the number of players playing is odd. Still, the change required is not colossal by any means.

Making a change to input syntax is quite simple to begin with. For example, if the syntax were to change from <rank><suit> to <suit><rank>, the only changes required would be the overloaded input and output operators and constructor with parameters in Card. The only other change required would be update the call to Card’s constructor in Deck’s constructor. Changing the command prompts is even easier. It will only require change to Human’s *makeMove()* method. Changing output messages to an invalid command can be done with relative ease too. The only place change might be necessary is in Human once again.

Changes to the strategies of Computer would only require changes in the *makeMove()* method respective to Computer. For example, if the last card or middle card were to be played or otherwise discarded, just two lines of code would require change in Computer’s implementation file. Changing strategies at runtime are discussed in greater detail under the heading, Answers to Questions.

Implementing an Observer or MVC pattern may require a significant amount of additional change, but very little change would be required of the preexisting code. The changes made would complement my program and design rather than contrast it. For example, Game may serve as the Model which inherits from Subject. Then, View would have to inherit from the Observer and additions may be made to the new classes. As explained, my design can have an Observer or MVC pattern implemented upon. This can then be extended such that it implements a graphical user interface.

An additional feature such as the history of moves or current scores may be implemented for individual players with relative ease. For example, if one were to introduce a new command, score, which would print the current Human player’s score, an addition of an if-else statement in Human’s *makeMove()* would fulfil the amendments necessary. To list the history of a Human player’s moves, an additional vector of two vectors of cards may be used in the Human class. Then, additions to Human’s makeMove*()* may be made such that it adds the cards played or discards whenever one is made to first and second array respectively. The final addition will constitute of an if-else statement in Human’s *makeMove()* to accommodate a new command, e.g. history that will denote the command to print the player’s played cards and discarded cards.

Lastly, hints may be provided to users based on the next players’ hands. This would have to be managed by Game since it has access to all the players’ hands. So, for example, if a player can play cards 7D and 7H, and the player after him will have no legal pays if the current player plays 7H, then the player may be notified to play 7H if it wished so. The same can be done with the Computer class. If so, every move a computer player makes will be based on the next player’s cards and if one human player was playing with multiple computer players, it would be practically impossible to win unless the human player used a hint every move. This could potentially lead to games with different difficulty levels whereby computers have set number of hints they can use. A limit to the number of hints may be introduced for Humans too. The Human's *makeMove()* method would have to handle such a specification as it re-prompts for input when the given input is invalid. To keep track of the number of hints used, an additional member field may be added to Human.

5 Final Questions

Note: The answer to the third question is the only answer that differs from DD1.

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: The design pattern I would use is the MVC design pattern. This would allow me to separate the implementation into three main components, Model, View, and Controller. I would have to implement 5 additional classes, namely, Model, View, Controller, Observer and Subject. I would most likely have to add a 6th class such as Xwindow in assignment 4, question 3 to display the graphical user interface. I have not done so as it would add significant complexity to the overall code and the implementation of a graphical user interface is not mandatory.

My code fits this framework as the Game class may be replaced with the Model class which inherits from the Subject class. Then, the View class would have to inherit from the Observer class and final adjustments may be made for my framework to fit the MVC design pattern.

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: Not much. Additional strategies may be implemented using additional classes that inherit from my Player class. As a computer players strategies change dynamically, I could copy the contents of the current computer player to a new computer player with the desired play strategy followed by the destruction of the old computer player.

Another implementation option may be to use a Strategy design pattern where the Computer class has an abstract Strategy class. Subsequently, the different strategies are implemented via individual classes, all of which inherit from the Strategy class. This would enable a computer player to make frequent strategy changes during runtime.

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: My design would remain relatively constant. The card multiplicity of the Deck would increase by 2 to 54. Other than the change in multiplicity, most of the remaining changes are to do with the member methods. Certain member methods such as those responsible for finding all possible legal moves and dealing the cards to the players will have to accommodate the change in rules in their implementations. For example, two players will be dealt 14 cards while the other two will be dealt only 13. Moreover, rounds would end with two players having a single card each in their respective hands and so on. All the above changes would be minor and easily implemented.

6 Final Questions

(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?

Answer: The most important think I learnt is to start early. Additionally, I have understood the significance of planning and designing a UML before starting to write object-oriented code as this helps minimize coupling and maximize cohesion. Test often, and compile and test the member methods of a class before moving on to the next one. This helps locate errors and makes bug fixing significantly easier at the end.

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

Answer: I would not spend the amount of time that I did thinking about design but instead journal my design choices and note the reasoning behind each choice. This would help me keep track of my though process and as a result aid in better design. I would try to think ahead and figure what kind of member methods I might require when implementing a new class. Additionally, I would not fixate myself on design as much as I did and focus on getting an executable ready first. I may implement an additional class that handles all the output messages or otherwise implement an MVC pattern to accommodate a GUI interface.