Final Project DD2: Straight Design
Chris He(20845382)

Introduction:

This is the design of Straight project for 1211 cs246 2021 winter final project.
This project is design and implemented by Chris He(20845382).
This project is done by myself. The reason I used "we" as the first-person pronouns is that this is design is also provided to whoever want to follow my design and implement this program, so I used "we".

Overview:

1. In my implementation, I have card.h, card.cc, player.cc, player.h, strategy.h, strategy.cc, deck.cc, deck.h, and main.cc as the file set, which allows me to have low coupling; meanwhile, these files have high cohesion. They are specifically for each important implementation part of my program such as card, player, deck, strategy and main file. This can allows me to have minimal recompilation for changes.
2. Firstly, I have (virtual and concrete) deck factory class, and (virtual and concrete) deck class that form a factory design pattern. Secondly, I have a virtual card class, a concrete straightCard class that form a template design pattern. Lastly, I have (virtual and concrete) player class, virtual strategy class, a concrete computer strategy class, and a concrete human strategy class that form a strategy design pattern and template design pattern.
3. Lastly, the relation of each classes is important. Notice, that card class is composited in player and deck classes. Player class is composited in deck class, and strategy class is aggregated in player class. When building the deck to start a new game, we call the buildDeck(shared_prt<Card> indexCardType, shared_prt<Card> indexPlayerType, shared_prt<Card> indexDeckType, std::vector<shared_prt<Card>> indexStrategyType). The variate indicate what types of player, cards and deck is needed for this game.

Also, the strategy class set will handle the any changes of strategy of player during the game by its method setStrategy(shared_prt<strategy> newStrategy).

Design:

1. Our implementation used three design patterns: factory, strategy and template design patterns.

   To create deck with different seed. We use factory design pattern. This allows me to have different types of deck to fit various game rules. We use the factory (deckBuilder) to build deck, so when we modify the game rules, the main() function and user interface would not have to be changed.
   To let players can be human or computer or even with other complex strategy to play the game, I used strategy design pattern to implement this. This allows me to easily modify player's strategy with either human, computer, or smarter computer strategy for "ragequit" and other command.
   To handle the possibility of having more cards, deck, strategy, player types, the deck, player, cards and strategy all have template design pattern. Because I build the program to use these concrete classes as a template process. If we build a new deck, player, cards or strategy, we can simply switch the new classes into the template process without any other editing of that original program other than building a new concrete class. This feature combined with different files strategy, my program would have less recompilation and modification. Also, template of strategy allows us to have human and different computer strategies at the same time, which is provided for the strategy design pattern of player class to change strategies frequently during the game.
2. In order to avoid memory leak, we use smart pointer(shared_ptr). The smart pointer helps us to handle the memory management without any other effort. However, we just need to make sure there is no loop smart pointer that would cause smart pointer fails to manage some objects' memory leak.
3. Our functions will be strong guarantee, this allows the program to handle invalid input and still run correctly without any impact. To have this feature, I use the try and catch exception for the command input from users. If the input is not valid or the command input is not a legal play for the player, the program will catch the exception and print the error message, and loop this process to retry the new command input until the program received a valid input. Only after the valid and legal command input, the program would modify the existing data.

Design Details' explanations:

1. Cards' class: We have virtual Card and concrete straight Card classes to store the cards information. We can use printCard() function to print out the cards. This allows me to switch types of cards with little impact to my structure. We can even have more cards types such as UNO. The Concrete straightCard class have two character fields to store cards. For example, "AS" has 'A' 'S' characters to stands for ace in spade, "3S" has '3' 'S' characters to stand for 3 of spades. When comes to comparing cards, we can have a friends helper function to take the first character field in each card and compare the character in different cases. For example, one case for "King, Queen, and Ace" we need to convert them into numbers "13, 12 and 1", and then compare by characters '1< '3' < '4'< '12'< '13'. We can override the = and < functions for cards class.
2. Player class: We have virtual player and concrete straightPlayer classes.
   Player has a vector of shared_ptr cards as discards list, which is to store the discards cards list.
   (aggregation) Player has a vector of shared_ptr cards for cards in hand.(aggregation)
   Most importantly, we have composition of strategy for player to make move. They might be human strategy, computer strategy or new smarter computer strategy.
   Also, we have a Boolean value to indicate if this player a human or computer, which can help us to debug.
   We have Int score to remember the old score of this player.

   Then we have setStrategy(Strategy & strategy) method for player to set strategy or switch strategies.

Move (Deck &deck) method is for player to make move on deck by taking reference of the deck.

printDiscards() and printScore() are simple functions to print the vector of discards and scores. We implement these functions here to simplify the move function in strategy, such that move method in strategy class can just call the print without implement its own print function or helper function.

3. Deck class: We have a set of classes to form a factory design pattern: virtual Deck, concrete straightDeck, virtual deckBuilder and concrete straightDeckBuilder classes.
   Objects created by the factory "deckBuilder" are virtual Deck and concrete straightDeck class.
   Straight Deck has private field int seed, vector of shared_ptr card for full cards sets with the order of seed's shuffling, and 4 vectors of shared_ptr cards for spades, heart, club, dimonds to store the played cards on the deck. Also, there is a temp vector of shared_ptr cards that stores new shuffling for starting a new round.
   Relatively, Deck class has printDeck() to print out current cards on deck, and printFullDeck() to print the full set of cards in this game with the shuffling order decided by seed.
   Also, it has a queue of shared_ptr players so we can easily keep track of the order of each players.
   Relatively, we have playMove() to instruct the current play to do a move on the deck with either human strategy or computer strategy.
   NewGame() method is for starting a new round while keeping players' score and strategy in last round. We set the temp vector of full cards set with the new order of shuffling. Then, we reset each player's vectors of 13 cards in hand with the new temp vector of full cards set. We will also set the players' discards list and vector of cards on deck to be empty.

   Deck or straightDeck classes have composition relationship of players, when game ends, deck deleted and players' information is deleted.
   Also, Deck or straightDeck classes aggregate cards for the deck, temp vector of 52 cards and the seed's shuffling vector of 52 cards on stack memory.

   These form a factory design pattern with deckBuilder.

4. deckBuilder class: We have virtual deckBuilder and concrete straightDeckBuilder classes as factory to create decks.
   straightDeckBuilder has private field seed, which can make sure deck is shuffled with the same order when having the same seed. We get the seed when execute the executable file and receive one argument as the seed. We create the straightDeckBuilder with constructor with the seed.
   When creating the deck, we would also create a vector of 52 shared_ptr cards with seed's shuffling, a vector of 52 shared_ptr cards with random shuffling for new round, and a queue of shared_ptr players as its fields.

   We will ask users to input whether each player is human or computer at first. Players are created with delivered 13 cards with the order of random shuffled 52 cards. We also need to check which player get 7 of spades. We need to set the queue of shared_ptr players with relative order. 4 after 3 after 2 after 1, and who has 7 of spades is placed at the front.
   The actual deck of Deck class is set to be empty at first.

5. Strategy class: Virtual Strategy and concrete humanStrategy and computer Strategy are the classes composited in player classes. These classes form strategy design pattern with players, which allows players have flexible strategy or methods to make a move.
   Strategy classes also form a template design pattern where we have a virtual root class strategy, and we can have many concrete strategies for human and computers.

   We set the strategy class as a friend class of player and deck classes, then we can let strategy class's methods to modify private or protected field of players and deck.
   If the player is a human, we print the cards of the deck on the table first. Secondly, we compare the vector of cards in hand and on the deck to find and print the legal plays. Thirdly, the program read input from users to make a move and check if that is a valid move. By checking the vector of cards on deck and vector of cards in hand, we can decide if it is valid move. If it is not a valid move then print warning message and make no changes. If it is valid, we modify the player and deck classes according to the users' input. We

modify discards list of the players, or the vector of shared_ptr cards in hand and the vectors of shared_ptr cards for deck according to discard move or play move. We print the discard or the play message after. Notice, we need throw and catch exception to handle the invalid command cases, in order to have strong guarantee.
For the third step, we might receive a ragequit command. We will call the setStrategy(Strategy &strategy) function to modify the current players' strategy to the computer one. Then, we execute move(Deck &deck) function again for the current player to make a move again.
For the third step, we might receive a deck command. We have the reference of our concrete deck and we can call its public method printFullDeck() to print the vector of all cards of the deck with the order of seed's shuffling.
Fourthly, if the final result caused the vector of shared_ptr cards in hand to be empty and someone's score exceed 80, the game ends. We would print the players' discards list, old score and new score by summing up his discards in this game, which follows the order player4 after player3 after player2 after player1. Then, we would compare which player has the lowest score and print the winning message of this player.
If no player has score exceed80, we would sum up the old score of each player and their score in this round to be new value stored in their score field.

6. Main function: In the end we need to create main.cc file that connects all the header files and its classes. In the main function, we take one input argument int as the seed. Firstly, we need one straightDeckBuilder created by its constructor taking the int seed. Secondly, we need a vector of shared_ptr cards, and the cards are created by cards' constructor. Then we need to create human strategy, simple, and smart computer strategy. Lastly, we need straightDeck created by the straightDeckBuilder; meanwhile, we would also build the players by the constructor of players.
   After getting the fields, we have a loop to call the move() method in straightDeck. By the design of move() methods in player, deck and strategy methods, we would interact with users' input and run the game routine. If we meet the end of one round and someone exceed the 80 points, our function will print the winning message and break the loop. Eventually, we would exit the program with strong guarantee according to the design of our program.

Resilience to Change:

Firstly, minimal modification:

1. If we want to have different types of deck for different straight game rules, we just need to modify the factory (deck creator) and create a new deck class type without changing other classes. It is because I used factory design pattern, so users and other functions don't need to know and use the details of the factory (deck builder). They just need to call the public method in factory and deck class. So, no impact to have new deck types. For example, if we want to have a deck that have various number of players and who get club 7 plays first, we just need to modify factory class (deck builder). Then, we can call the creator method in factory class (deck builder) to create a new specific deck shared_ptr for the program.
2. If we want to have more strategy types, we can create a new strategy and only modify the concrete player class to use the new strategy. Because of the template design pattern and strategy design pattern, I can have various strategy and have high flexibility to switch the strategy during the game; meanwhile, these design patterns allow me to only change 1 class to implement this new feature.

3. If we want to have more player types, cards types and deck types, we can add more concrete player, cards and deck classes and modify the factory (deck builder) to implement this feature. Because of the template design patter, as long as I keep the public method to be the same, there would not be any impact to other classes. Also, my creator method in the factory (deck builder) need to take input to indicate which types of deck, player and cards are needed. We can just specify a new types index number for each new deck, player and cards types to supports the possibility of various changes to these classes' specification, which is supported by the template design pattern and factory design pattern. Only one small changing of concrete factory class and creating one new concreated class is needed for this modify of the program.

Secondly, minimal recompilation:

1. I have card.h, card.cc, player.cc, player.h, strategy.h, strategy.cc, deck.cc, deck.h, and main.cc as the file set. We have high cohesion in these files because they are specified for only one token of function in my program. card.h, card.cc are for the cards in the program, so if we want to have new card type, we just need to modify this single set of files. Also, the other functions in other files only use the public printCard() in in card.cc card.h, and they only composite the virtual Card class. So, other classes can composited any new and various concrete card types without modify the classes, methods, and function in player.cc, player.h, strategy.h, strategy.cc, deck.cc, deck.h. They only call the same public method in card.h. Therefore, ss long as the public method of new concrete card class stay the same, we would not need to change or recompile other files, such as player.o(player.cc, player.h), strategy.o(strategy.h, strategy.cc) and deck.o(deck.cc, deck.h) and main.o(main.cc, player.h …). We only need to recompile cards.o(cards.cc, cards.h).

Thirdly, different cases of changes:

1. I explained if we have new or different cards type. We just create or change cards classes in cards.h and cards.cc. If we let the cards class have a new field that record the brand of the cards set such as "Air Jordan Nike collaboration". We just modify the concrete class type and its printCard() method to print new brand's field. Only one file changes and one file recompilation. If we want UNO cards set, we just create a new concrete cards set named as UNOCards. In order to let our program use the new UNO cards type. We can modify the main file, when calling the buildDeck(shared_prt<Card> indexCardType, shared_prt<Card> indexPlayerType, shared_prt<Card> indexDeckType, std::vector<shared_prt<Card>> indexStrategyType) method in the factory(deckBuilder) classes, we use the UNOcard class variate as indexCardType). Then, the factory would build a new game with the new UNO cards type. In this case, we need to modify and recompile two files: main.cc and cards.o(cards.h, careds.cc). Or, we can let he int main() function to get argument in the terminal command line. The argument would specify the new UNO cards tyeps. When main function call the buildDeck(shared_prt<Card> indexCardType, shared_prt<Card> indexPlayerType, shared_prt<Card> indexDeckType, std::vector<shared_prt<Card>> indexStrategyType) method in the factory(deckBuilder) classes to use the new set of cards types from the argument in the command line. In this case, we only need to recompile and modify the cards.o(cards.h, careds.cc). However, this case will change the complication rules of the original program. Notice, the public method of UNO cards class need to be the same as the straightCard class, because other methods in other files will call these same public method to print or create new cards.
2. If we want to have different player and deck types, we only need to modify and recompile these clases in their own specific files. They have public methods for making a move, printing cards in hand, and printing cards on deck. Other files would only call these public methods. So, we can just modify their implementations to handle new rules of game. For example, if we need the player to have double strategy, such as human and computer strategy: the program will provide the computer strategy move as suggestion and let the human strategy decide the actual move. We just need to modify the fields of player class with one more strategy, and when making a move, we print the suggestion of computer strategy first and then let human strategy make a move. In this case we only need to modify one file and recompile one file.
3. If we want to have more player and deck types, we just need to create a new concrete class types in their own file. Then, modify the main file to use the new player or deck types. Also, we can use the second way of adding cards types, we can modify the way that main file taking the compiling commands to take the player and deck type specification.
4. Lastly, it's for the changes of strategy. Notice, when a player calles makeMove(xxx) method to make a move, the program would call its strategy to make a move. Also, the strategy field of a player is virtual strategy class, so we can have different concrete strategy types by just letting main file to create deck with different set of strategy as I discussed above. If we want more strategy, we just create a new concrete

strategy type in strategy file or modify a concrete strategy class. Because other methods in other type of classes would only call the public method strategyMakeMove(xxx), we can implement any strategyMakeMove(xxx)method as I want without affecting other files. Also, I we want to change the strategy dynamically during the game, this feature can be composed inside the strategyMakeMove(xxx) method.

Hence, these are the all-possible ways to change the program, each case is handled by either template, strategy or factory design patterns. The low coupling and high cohesion idea of my program helps us to handle the changes easier. Therefore, any changes will only cause small impact, minimal modification and minimal recompilation to our program.

Questions:

1. **Question: What sort of class design or design pattern should you use to structure your game classes so that changing the user interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this**

   Our implementation used three design patterns: template, factory and strategy design pattern.
   To create deck with different seed or with different rules for different games, we used factory design pattern. It has a virtual deck builder and we can implement various concrete deck builder. This means we can have different builder to build straight game with different rules or user interface by just creating a new builder without modify the old builder, which is flexible. For example, we can have a deckBuilder that create a deck with deck fields that contain jokers.
   We use template design pattern for player classes. This allows us to have flexibility to build new kind of players without modify the old player types. For example, an observer player.
   We use strategy design pattern for player classes, so we can have flexibility to change players playing rules either human, computer or other game rules. For example, we can change the rules that player can discard while having valid move by just creating a new human strategy, without modify the old strategies.
   We kind of have template design pattern for straightDeck, we can have different deck types, such as a deck type that doesn't have spades.
   We used template design pattern for the card's classes. We have virtual and concrete card's classes. So, we can even change what type of cards are we using. (poker cards, UNO, or others)

2. **Question: If you want to allow computer players, in addition to human players, how might you structure your classes? 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 structure?**

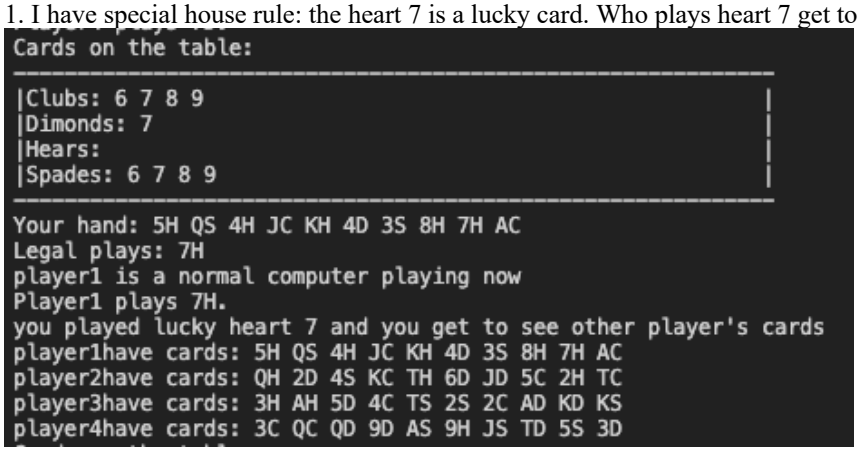
   We used strategy design pattern. So, we have the flexibility of the ways for each player to make moves. We have my concrete straight players and this class aggregate strategy classes. We use template design pattern for strategy class, so we can have human strategy, computer x simple strategy and computer smart strategy. If someone calls rageout, we would call setStrategy method of players to modify the player's strategy field to be computer strategy. In order to dynamically change the computer strategy. We can check if a player is a computer and he is losing too much (relatively high scores), we can change computer strategy to the smart computer strategy, vice versa for changing into simple computer strategy. These changes would not affect my structure too much. We can even add one more computer strategy by just adding a new concrete strategy class without modify the existing player and strategy classes. Hence, very little changes can fulfill the needs of changing strategy(computer, human, or automatically change to smart computer when needed).

3. **Question: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?**

   As we mentioned in the last question, we have different strategies types and strategy is aggregated in player class. They form the strategy design pattern. When we have rageout command, we would call setStrategy method of players to modify the player's strategy field to be computer strategy, and from now the current playing player will have the computer strategy.
   Here are the details of the implementation: When doing one human strategy move and we receive a command of ragequit, we will call the players' setStrategy(Strategy &strategy) function to change the strategy of the current player. Then, the player make a move with the new computer strategy by calling the move (Deck &deck) method again with the computer strategy.

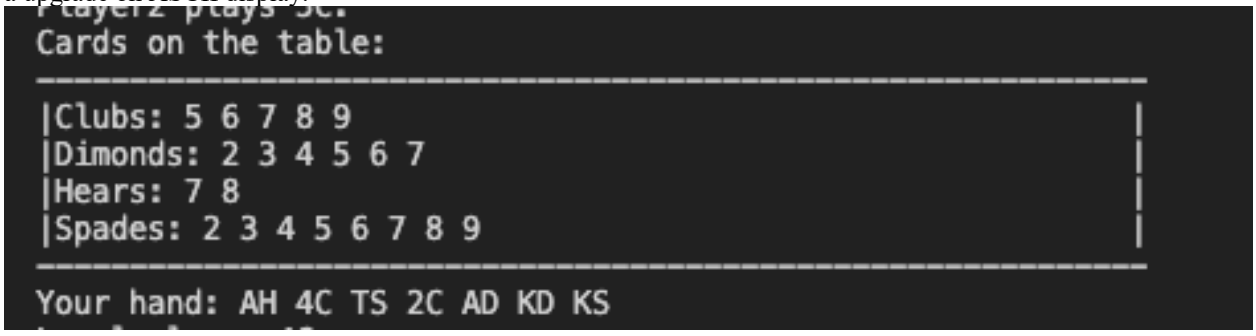
Extra features:

1. I have special house rule: the heart 7 is a lucky card. Who plays heart 7 get to see other player's cards in hand.



2. My program has normal computer and smart computer. If the player is having the highest score, the smart computer would discard the lowest rank's card, and play the highest rank's card. Otherwise, it is a normal computer. The normal computer player discard the first card in hand, If there is no legal play. Also, the normal computer plays the first legal play card in hand, if there is legal play.

3. In order to have a better view of the game interface, this program have a special display of deck. This is feature is a upgrade on ASCII display.



Final Quesitons:

1. I worked alone, so I realize it is more important to plan the time well, because there is no teammate who checks you to work step by step. The schedule is the only thing that pushes you to work forward. A smart plan should be feasible and encouraging. During this project, my plan is too hard finish on time, because I expect the implementation time too short. After finishing this project, I can use the experience from this project to help me building a better, feasible, encouraging plan of attack in the future.

2. Next time, I would start implementing the classes that have low dependency on other classes first. Because only when I start to write the code, I would understand what is the feasible design for each program. So, I often modify the structure of my classes during the implementation, which caused a big problem for me. For example, I created the deck and player classes first, and then is the strategy class. When I writing the strategy class, I notice the strategy class is hard to reach some field in the player or deck classes. So, I was adding methods to player and deck classes during the implementation of strategy class. Modifying classes is normally slower than building a new class, because I have to recheck the design of the deck and player classes again to make sure the new method would not impact other stuff in the method.

Conclusion:

This is not a hard project on algorithm, but it is more important to implement and design this project to be flexible and clean. Different classes and files with high cohesion and low coupling helped this program to be flexible. It is very easy and highly possible to add new feature to this program and make it more interesting and powerful.
This is the plan of the straight program. Best wish to the developers who is implementing this program following this plan. Have a great day and have fun!

This the original plan of attack:

Plan of Attack:

1. Firstly, we create all the different header file and cc file that contains different functions of my program. According to my UML, we need one set of files for Card class, one set for Strategy classes, one set for deckBuilder, Deck and straightDeck class because they form a factory design pattern, and then one set for Player and straightPlayer class. At the end, we need a mian.cc file to connect these separating function sections. This takes 15~20mins (on April 12)
2. Third step is to implement card class for 15 ~ 20-mins (on April 12).
   And use same amount of time to debug individually.
   This is a simple individual class to store card, so it is implemented at front. The sequence of implementation follows the rule that is from simple to complex, independent to high dependency.
3. Then is to implement player class for 15~20 mins (on April 12).
   Then use same amount of time to debug individually.
   It is an intermediate class with no complex functions.

4. Implement Player and straightPlayer class for 1 ~1.2 hrs (on April 12th).
   And use half of the implementation time or even more time to debug. (or less)
5. Implement Deck class and straightDeck class for 1~1.2 hrs (on April 12th).
   And use half of the implementation time or even more time to debug. .(or less)
   This is harder and after player class implementation because it composes player classes.
6. Implement deckBuilder and straightDeckBuilder class for 1~1.2hrs (on April 12th). And use half of the implementation time or even more time to debug.(or less)
   This is after Deck class because deck builder is the factory to build deck.
7. Implement Strategy, humanStrategy and computerStrategy for class for 1 ~2.2 hrs (on April 13th). And use half of the implementation time or even more time to debug. (or less)
   The reason to place strategy implementation at the end is because this is the actual part of function to modify the deck and player class following the rule of game. This is the most complicated so we implement this at the end
8. Then, we implement the main.cc file to connect all the functions for 1~2.2 hrs(on April 13th)
9. Because of the individual debugging strategy, it should take around 5 hrs to debug for the total program, and for the right functionality. (on April 13th)

It should take about 2~3 days to finish this straight project.

This the new plan of attack after the implementation:
(This is provided for the people who want to follow this design to implement as a reference)

Plan of Attack:

1. Firstly, we create all the different header file and cc file that contains different functions of my program. According to my UML, we need one set of files for Card class, one set for Strategy classes, one set for deckBuilder, Deck and straightDeck class because they form a factory design pattern, and then one set for Player and straightPlayer class. At the end, we need a mian.cc file to connect these separating function sections. This takes 15~20mins (on April 12)
2. Third step is to implement card class for 15 ~ 20-mins (on April 12).
   And use same amount of time to debug individually.
   This is a simple individual class to store card, so it is implemented at front. The sequence of implementation follows the rule that is from simple to complex, independent to high dependency.
3. Then is to implement player class for 15~20 mins (on April 12).
   Then use same amount of time to debug individually.
   It is an intermediate class with no complex functions.
4. Implement Player and straightPlayer class for 1 ~1.2 hrs (on April 12th).
   And use half of the implementation time or even more time to debug. (or less)
5. Implement Deck class and straightDeck class for 1~1.2 hrs (on April 13th).
   And use half of the implementation time or even more time to debug. .(or less)
   This is harder and after player class implementation because it composes player classes.
6. Implement deckBuilder and straightDeckBuilder class for 2.2hrs (on April 14th). And use half of the implementation time or even more time to debug.(or less)
   This is after Deck class because deck builder is the factory to build deck.
7. Implement Strategy, humanStrategy and computerStrategy for class for 2.2 hrs (on April 14th). And use half of the implementation time or even more time to debug. (or less)
   The reason to place strategy implementation at the end is because this is the actual part of function to modify the deck and player class following the rule of game. This is the most complicated so we implement this at the end
8. Then, we implement the main.cc file to connect all the functions for 2.2 hrs(on April 15th)
9. Because of the individual debugging strategy, it should take around 6 hrs to debug for the total program, and for the right functionality. (on April 15th)

It should take about 2~4 days to finish this straight project, my schedule is lagged because of my other course work.