The structure of my project is based on the following classes: Card, Deck, Player, Table, Observer, Subject, AddText, GameState, and GameController.

Card Class:
The Card class is the foundation of this project and it is composed of the following methods:

+ Card(Suit suit, Rank rank) : constructor
+ Card() : constructor
+ getSuit() const : Integer // accessor
+ getRank() const : Integer // accessor
+ printCard() const : String

I used enum to keep track of the suits and ranks of cards. The printCard() method returns a string that displays the card with proper syntax (i.e., <rank><suit>, such as 7C). The Card class is the constituent part class that creates the Deck class. The card class differs from Due Date 1. Instead of using a Char for suit and integer for rank, I used enum for both.

Deck Class:
The Deck class is responsible for creating the Deck that players will be using for the game. It is composed of the following methods:

+ Deck(rng : default_random_engine) : constructor
+ createDeck() : void
+ shuffle() : void
+ getCard(i : Integer) const : shared_pointer<Card> // accessor
+ printDeck() : void

The createDeck() method is responsible for creating the deck of cards at the beginning of every game prior to shuffling. createDeck() creates the deck in the following order:

```
AC 2C 3C ... TC JC QC KC AD 2D ... QD KD AH 2H ... QH KH AS 2S ... QS KS
```

. The shuffle() method modifies the deck by shuffling the deck once and storing the new order of cards as the deck. Finally, printDeck() displays the deck to standard output by printing each card of the deck in order. This method will be called when a human player inputs the “deck” command while playing the game.

Player Class:
The player class creates the players with a default value set to human player (“h”). However, this value is immediately changed to computer player (“c”) if a computer player is selected at the start of the game. Player objects that are created during runtime are unique in terms of the cards that each player possesses, their discard pile, their player number, and their score. However, each Player points to the same Table that they all play on. Player is composed of the following methods:

+ Player(table : shared_pointer<Table> table , playerNum : Integer) : constructor
+ ~Player() : destructor
+ addCard(card : shared_pointer<Card>) : void
+ findCard(card : Card) const : Integer
+ discard (card : shared_pointer<Card>) : void
+ playCard(Card : shared_pointer<Card>) : void
+ startingPlayer() const : Boolean
+ isHandEmpty() const : Boolean

+ getScore() const : Integer // accessor
+ getOldScore() : Integer // accessor
+ getDiscardPileScore() : Integer
+ updateScore() : void
+ getPlayerNum() const : Integer // accessor
+ clearHand() : void
+ clearDiscardPile() : void
+ getFirstCard() : shared_pointer<Card>
+ getFirstLegalPlay() : shared_pointer<Card>
+ getNumDiscards() const : Integer
+ getNumCardsinHand() const : Integer
+ getNumLegalPlays() const : Integer
+ setLegalPlays() : void
+ getHand() : vector<shared_pointer<Card>> // accessor
+ getDiscardPile() : vector<shared_pointer<Card>> // accessor
+ getLegalPlays() : vector<shared_pointer<card>> // accessor
+ getTable() : shared_pointer<Table> // accessor
+ getCard (i : Integer) const : shared_pointer<Card>
+ becomeComputerPlayer() : void
+ getType() const : String

addCard() is responsible for adding cards from the deck to the Player's hand. This is because, at the start of each round, 13 cards are added to each player's hand. playCard() and discard() are responsible for handling the player's moves (play card / discard card) during runtime. After calling either method, the card that the Player chose to play or discard is removed from their hand. startingPlayer() is used to determine which Player has the 7 of Spades at the beginning of each round. isHandEmpty() determines whether the round has been finished. findCard() determines whether the Player has a specific card in their hand by returning an index value to the card in their hand. If the card was not found, then a negative integer is returned. getDiscardPileScore() both updates the Player's discardPileScore and returns the sum of the ranks of all the cards in the Player's discard pile. updateScore() is used at the end of each round to calculate the total score of the Player. This is because if the Player's score is 80 or above, the game ends. setLegalPlays() looks at the table and returns all of the Player's updated legal plays.

The Player class was one of the classes that changed from Due Date 1. Initially, I had planned to create subclasses HumanPlayer and ComputerPlayer that inherits from the abstract base class Player. However, since both subclasses have identical methods and implementation with the only differing feature being playerType, combining the three classes into one class was more efficient. This had little to no impact on the cohesion and coupling of the project as well.

Table Class:
The table class is responsible for keeping track of the cards that have been played by implementing a map. The map keeps track of each of the four suits and the corresponding cards that have been played. Analogously, the map creates four piles, each corresponding to a different suit, on the table. Table has the follwoing methods:

+ Table() : constructor
+ addCard(suit : Suit &, card : shared_pointer<Card>) : void
+ discard() : void
+ clearTable() : void
+ isLegalPlay(card : shared_pointer<Card>) : Boolean

+ isTableEmpty() : Boolean
+ getPile(suit : Suit) : vector<shared_pointer<Card>>

Table is the concrete subject component of the Observer Design Pattern that I implemented in this project. It inherits from the Subject class and whenever a Player adds or discards a card, the Table class calls notifyObservers(). The addCard() method places the card that the Player plays onto the correct pile given that the card being played is legal. The purpose of the discard() method is solely for calling notifyObservers() when a player discards a card. isLegalPlay() determines whether it is legal to play a specific card being passed in as an argument. clearTable() is used prior to each new round and it is responsible for creating a new set of empty piles. getPile() returns all of the cards in the pile corresponding to the suit being passed in as an argument.

Subject class and Observer class:
These classes are the Subject and Observer classes in the Observer Design Pattern.

Subject methods:
+ attach(o : Observer *) : void
+ detach(o : Observer *) : void
+ notifyObservers() : void
+ *~Subject() : destructor*

Observer methods:
+ *notify() : void*
+ *~Observer() : destructor*

AddText class:
AddText is the concrete observer class that attaches itself to the Table class. Its notify() method prints out the cards that are currently on the Table. It inherits from the Observer class.

GameState class:
GameState keeps track of the status of the game. It determines whether the game is over, whether a round is over, the current player's turn, and the maximum score among all players. It has the following methods:

+ GameState() : constructor
+ ~GameState() : destructor
+ setPlayers(listofplayers : vector<shared_pointer<Player>> &) : void
+ setMaxScore() : void
+ setCurrentPlayer() : void
+ getCurrentPlayer() const : Integer
+ isGameOver() const : Boolean
+ isRoundOver() const : Boolean
+ setGameOver() : void
+ setRoundOver() : void
+ getPlayer(playerNum : Integer) : shared_pointer<Player>
+ getWinners() const : vector<Integer>
+ rageQuit(currplayer : Integer) : void
+ reset() : void
+ setRoundStart() : void
+ endGame() : void

setMaxScore() is called at the end of every round to determine whether or not the game should end. setCurrentPlayer() determines which player's turn it is. setGameOver() sets the member variable isGameOver to true if the round is over and the maximum score has reached or surpassed 80. Similarly, setRoundOver() sets the member variable isRoundOver to true if every player in the game has an empty hand. getWinners() is called at the end of every game to determine which Players have won. rageQuit() is called if a human player decides to ragequit, and it sets their playerType to computer player. reset() is called at the start of every new round to set the variable isRoundOver to false.

GameState is another class that changed from Due Date 1. Initially, I had planned for the GameState class to be the concrete subject. However, I realized that it makes more sense for Table to be the concrete subject since the table is constantly being updated and changed. The observers should be notified every time a new card is played or discarded so that the next player can see what cards are on the table. It no longer inherits from Subject.

GameController class:
GameController is the object that is created in main.cc and is responsible for running the game. It has the following methods:

+ GameController(rng : default_random_engine) : constructor
+ getTable() : shared_pointer<Table>
+ getDeck() : shared_pointer<Deck>
+ getGameState() : shared_pointer<GameState>
+ getPlayers() : vector<shared_pointer<Player>>
+ startGame() : void
+ invitePlayers() : void
+ dealCards() : void
+ startRound() : void
+ playRound() : void
+ printEndRoundResults() : void
+ printWinners() : void
+ printHand(playerNum : Integer) : void
+ printLegalPlays(playerNum : Integer) : void

startGame() creates the deck of cards, creates the table, and creates an instance of GameState. invitePlayers() asks for user input at the start of the game to initialize each of the four players. dealCards() assigns 13 cards to each player. startRound() starts a new round by clearing the Table, resetting the game state, clearing the players' hands, shuffling the deck, and redealing the cards. playRound() handles all the logic of playing an entire round where each player takes turns until all players have no cards in their hands. printEndRoundResults() prints the scores of each player and finally, printWinners() prints the players that won the game.

GameController was another class that changed from Due Date 1. I added a few more methods such as invitePlayers(), dealCards(), and playRound().

I used an Observer Design pattern in this project in order for it to be able to accommodate any change in the specifications with high efficiency. For example, if we wish to change the interface from text-based to graphical, I would only have to make changes to the concrete observer. Additionally, changing the game rules would only impact my GameState class and GameController class. This is a result of high coupling and low cohesion. Each class is fairly independent of one another. For example, the Player class only handles everything related to Player, and the Card class handles everything related to cards. As a result, changing something major in one class would not affect any other classes. For instance, adding Jokers to the deck or changing the number of cards in a deck would have an incredibly small impact on classes other than Deck and Cards. The only other method that would need changing is dealCards() in the GameController class, where instead of dealing 52 cards, 54 cards would have to be dealt. Furthermore, changing the rules of the game would only affect GameState and GameController. The level that my project conforms to object oriented programming can be seen in the main.cc file. There is very little code, and the entire program is run by calling methods in the GameController class.

There are fairly few changes to the UML from Due Date 1. The major change is that GameState and Table have been swapped, and GameState is no longer a subclass of Subject. I also removed the Command, HumanPlayer, and ComputerPlayer classes. Other changes include updates to the attributes and operations of classes.

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.

The Observer Design Pattern will allow changing the user interface from text-based to graphical or changing the game rules, to have minimal impact on the code. Similar to the concepts presented in A4, such as the addtext observer and addgraphics observer, changing the user interface only requires the programmer to make changes to the notify method. This is because the logic of the game remains unchanged. Changing the game rules will affect the “subject”, which is what the observers are attached to. The notify methods will then remain unchanged while only the concrete subject classes and other related classes that affect the logic of the program will change. My classes Table, Subject, Observer, and AddText fit this framework and it is shown above.

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?

A Decorator design pattern should be used in this case. The Decorator design pattern allows the programmer to add functionalities, changes, or features to an object at run-time rather than to the entire class as a whole. If there is a possibility that different types of computer players might have differing play strategies, or if their strategies might change during the play of the game, it would be wise to implement a Decorator design pattern in addition to the Observer design pattern. I would then need to implement a component class, a concrete component class, a decorator class, and concrete decorator classes. The game strategies would be the concrete decorator classes, and they would be added to the Player object since different players would have different play strategies. Of course, this would affect the UML diagram of the program. It would lead to more generalization/specialization association between superclass and subclasses as well as aggregation between the decorator and component.

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?

This would have a very small impact on the design of the program. Firstly, this would change the Card class. The Jokers would have different rank values and suits which must now be accounted for. Consequently, this will affect the design of the Deck class and how the cards would be dealt each round. Additionally, the logic behind legal plays and illegal plays would change. Players with a joker in their hand would have more legal play options. Additional methods would need to be defined that handle converting the joker into a specific card of the player’s choice. However, the overall class relationships would not change and independence between classes is still preserved.

There are no changes to my answers from Due Date 1.

Extra Credit Features:
An extra credit feature I included was implementing smart pointers. I did not have to manually clear any data and I had no memory leaks. This makes my code cleaner and shorter. Manually clearing data can easily result in memory leaks. By using shared pointers, I did not have to worry about forgetting to clear data in my program that would lead to memory leaks if left untouched.

What lessons did you learn about writing large programs?
A key lesson that I learned from writing large programs is the importance of starting small before extending. By starting small, I had a clear idea and goal of the next step that I had to take. For example, I started off by creating the Card class. Afterward, I extended the solution step by step by creating the Deck class, followed by the Table class, followed by the Player class, and so on. As a result, I had a perfectly clear understanding of the relationships between each and every class. Additionally, I was able to solve problems one by one. Newly discovered information and realizations could be applied while coding the next step, and this provided me with a smooth flow and clear path of approaching the program. Another lesson I learned was the importance of testing smaller portions of the program prior to the whole. Every time I completed a new class, I compiled to detect any errors in my code. As a result, by the time I completed every class, I only had to focus on logical errors rather than syntax. Finally, I learned about the importance of not procrastinating. By coding a little bit every day, I was able to complete this project sooner than I anticipated. By coding every day, the problem was always fresh in my head and debugging problems took substantially less time since I had more time to think about them.

What would you have done differently if you had the chance to start over?
I would have written unit tests prior to starting Due Date 2. If I had written tests prior to completing my code, I would have saved substantially less time testing my code. Instead, I manually tested my code after completing my project which took a lot of time. It also led to a higher chance that I would miss something crucial and forget to test a certain case.