Breakdown of Classes:

The game functions primarily through the following classes:

- Cards class
- Deck class
- Table class
- Player class
- Strategy class and its sub-classes

The implementation of each class is as follows:

Cards class:

The card class holds two variables. Both variables are an enumerated type labeled Suits and Rank. There is high cohesion in the Cards class as every method within the class only serves to perform operations with cards such as output, input and comparators along with getters. There is extremely little coupling within this class, only replying on the string and io libraries to function properly.

Deck Class:

The deck class stores a vector of Cards obtained through the Cards class. The methods within this class are the constructor for a deck which initialize the vector with all 52 cards of a deck, along with a shuffle method which shuffles the deck and another method to display the contents of a deck. There is also a getter method within this class to get the card located at a specific index of the deck. There is a moderate amount of coupling within this class with the Cards class as the existence of the deck relies completely on the Card class and the output of the deck relies on the output operator of the Cards class. The deck also uses many standard c++ libraries to function.

Table class

The table class functions by holding four vectors of cards. Each vector of cards represents a suit and the cards contained represent the “pile”. The methods within the class consist of constructors and a method to display the table. It also contains methods determining various information regarding the table such as whether a specific card can be played legally and if the table is empty or not. This information can then be used in the Player class to determine legal moves that the player can make. The other methods add cards to the table. This method relies on using the Card class’s getter methods to operate causing slight coupling between the Table class and the Cards class. The final method clears the table for a new game to begin.

Player Class:

The player class has a moderate amount of coupling with the other classes listed above. The class holds the player’s score, id (a number between 1-4) and shared pointers pointing to the table, deck and an instance to a Strategy class. The player also has a vector of Cards that act as the discard, another for the player’s hand and another to hold the player’s legal plays. All player-unique stored variables have a getter method for them or a way to display them. Some methods within the player class use the methods implemented in the other classes. Playing a card requires the method to add cards from the Table class, displaying the deck through the player class calls the deck displaying method from the deck class. Other than the getter and constructor methods, the player class uses mostly the vector library to function. Methods such as the play and discard_cards method uses the erase functionality from the vector library to remove cards from hand after the method was called. Shared pointers also play a large role within the Player class to remove the need to individual destructors for each other class that is created.

Strategy class and subclasses

The strategy class is highly coupled to the Player class. The single method located in the Strategy class relies heavily on the other methods found in the player class and the method within the strategy class require a reference to a player to be pass within it. The strategy class's main role is to take user input if the strategy algorithm is a Human's and to make appropriate plays using the Player class methods with the given inputs. If the algorithm is for a computer, the method simply plays cards if there are cards available to play and discards cards otherwise. All of these functions reference the methods used within the Player class and any changes to the Player class in terms of how the methods function will definitely cause the Strategy classes to need altering.

Straights.cc: The Main()

Within the main() method of the program, upon starting the program, a shared pointer of a Deck and Table are initiated onto the heap along with a vector of shared pointers of players. The program then loops four times asking the player to input whether a player is a human or computer. Depending on the results of the input, the program creates a shared pointer instance of a Human or computer strategy and creates a new instance of a player calling the constructor passing through the deck, table, id and strategy as parameters. The newly created instance is then pushed back to the player vector. After all the players have been created, the program shuffles the deck and distributes the first 13 cards to play 1, the next 13 cards to player 2 and so on. The game function on two while loops. The outer while loop checks if at least one player's points exceed 80. The inner while loop checks for the number cards in the player's cards. When the inner while loop terminates, the Players and table are reset, the deck are reshuffled and re-distributed. The game functions by first determining which player begins the game. The index of the starting player is saved to a variable and the while loop begins by calling the player located at the stored index. Each time the loop continues, that variable is increased. When the variable exceeds 3, the program sets the variable value back to 0. After a player gets 80 or more points, the program calculates the lowest point out of all the players and output the players with those points as the winner.

Changes:

There are many changes that this program currently supports.

If there are different commands that the players should be able to input, this can be completed by just changing the human strategy subclass and execute the inputted command as specified by the change. If there are different ways that the cards should be displayed, this can be achieved by just changing the output operator of the Cards class as every method that output the cards contained within them all call the Card output operator method.

Any changes to computer playstyle can also be done by just implementing a new strategy sub-class for each computer playstyle. This is helpful as the Strategy sub-classes are highly coupled to the Player classes hence, the player class would not have to be changed in order to program new computer-players as the computer sub-classes can in ways be seen to be an extension of the Player class.

My code is able to accommodate change extremely well as each main component of the game is directed to individual classes and the methods within those classes are strictly related to working only with the newly created class. Any need to change how a component functions within the program can be resolved by changing the appropriate class related to the change and no other changes should theoretically need to be made within the program UNLESS this change is a completely new feature that is not implemented previously by any means.

Most of the classes are very lowly coupled with each other so any large changes to any methods would lead to very small (if any) changes to the other methods in other classes.

Answers to the Questions revised:

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 classes that I created have very high Cohesion within the class. If there were any changes to any of the game such as how the deck should be displayed or how specific aspects of the game should be displayed, a slight change in one of the class's display methods will change that issue. If the game's rules were to change, all the game interactions are handled by a single Player class and hence, any changes to the rules can be solved by changing how the methods function within the player class. Of course, the design pattern I have chosen : Strategy handles how the player will interact with the game and by changing the functionality of the Strategy methods, the game rules can be changed accordingly as well.

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?

The answer to this question has not changed from my original. The strategy pattern allows for different algorithm to be used throughout the game and for the algorithm used by each player to change as the game progresses. If this only affects changing the computer algorithm, this can be set-up by having code within the computer strategy method determine appropriate strategies to use and to change the Player's strategy throughout the game when appropriate by simply creating a new instance of a specific algorithm and getting that specific player assigned to that algorithm. This is due to each player's round being completely reliant on calling the Strategy method assigned to those players.

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`?

Within the table class, there can be two card variables that hold the intended values of the Joker. In a common deck of cards, there's usually two types of Jokers and decks often differentiate these jokers so they can be used in different types of games. If the deck were to be set-up similarly such as there was a Joker variant 1 and Joker variant 2, the Table class can then hold a Card variable that associates to what Joker variant 1 and 2 represent and those variables can be referenced whenever a Joker is read from the table. Other design changes would be changing the output operator from the Cards class to be able to output the joker and to change the input operator to be able to read in the Joker card. The Deck class would also have to be changed so that the extra two jokers will be added into the deck on creation.

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?

Working with large programs definitely taught me the importance of planning things through especially with object-oriented programming where Classes become a very core aspect on program functionality. At the beginning my Deck and Table were not going to be classes. They were going to just be vectors that would be initialized upon program launch and there would be function within the main method to deal with outputting the Deck and table. This, however, became a very messy and hard to read process and made the code very disorganized. I then had to re-do a large part of my program so that it would function properly with my newly made Deck and Table classes. Another important lesson I learned is the importance of testing each class separately if possible. Being able to test whether the deck works during the creation of my deck class rather than after de-bugging after every class has been implemented definitely saved me the headache of reading multiple lines of compiler errors.

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

Organization comes to mind. Again, to recap my answer from above, I should have had thought more critically about how the program would be implemented and how the classes would interact with each other and what necessary methods I would need to create. There were often cases where I would finish a class and test the implementation of the class and realise when creating another class that coupled with that completed class that there were specific getter and function method that were essential in being able to complete the implementation of this new class. In other words I would have a better plan of attack and really draw out my game plan including all the methods I would need and how those methods can be created and what is needed from each component of my program to achieve that end goal.