Main User Stories

We designed our project, a poker game that can be played over a network connection, in the style of MVC, that is one where Model, View, and Controller are all separated and only interact with each other sparingly. We were able to break up the tasks in a way that we were able to work on several parts of this at once. To handle the over network side of the project, we simply divided the controller into two separate pieces, one for the client side, and another for the server side, but more on that later though.

For this project we have had several user stories that we used in order to develop a very highly functional game.

- *"As a programmer, I want the Model to be concise and implement classes to make the Model functional. These should include classes for the game, the players, and the different cards and their possible hands, and to implement any other classes in order to make a functional model."*

  We were able to achieve this through planning and next to flawless execution while writing our code. We created a Game class that runs through the actual game, going through the various turns of a card game.. The game class also has players that keep track of certain information for the player, such as their money and the best possible hands that they themselves can have. We also broke up the work between ourselves in concise ways that when following our own plan for the model the individual pieces fit together perfectly.

- *"As a programmer, I want the view to be well written, so that it is easy to modify in the future if need be, and I also want it to be implemented in a way that is both simple and clear to the user what is actually happening in the game and how they are to interact with the game."*

  These tasks were accomplished rather easily through the use of WindowBuilder than re-factoring of the code afterwards to clean up its layout. This resulted in code that both created an effective GUI and was itself very clean and easy to manipulate in the future if need be, such as when it was being hooked up to the controller.

- *"As a programmer, I want the controller to work well with both the model and the view so that it interacts with them without lowering its own cohesion, so that it does not overstep its boundaries."*

  We were able to achieve this by making the model and the view have high cohesion so that they were able to handle all functions without help from the controller, simply needing the command from the controller. We also set up the controller to handle much

of the controller internally. This helped us to keep our model free from changes due to networking needs.

- *"As a programmer, I want the hand to be able to implement a way to both calculate the best possible hand made from a group of cards, but also to be able to compare hands against each other to see which is the winner. In this way it should implement some form of hand logic, understanding poker to some degree."*

  We were able to implement this with a combination of classes, all housed as part of the model. We made a card utility that helped to analyze the cards that are available to each player and create their top hand by creating one of several hand objects, one class for each possible hand, i.e. flush, straight, etc. The implementation of this works rather smoothly as these hands are made comparable as they are sub-classes of the comparable Hand class. This allows for easy creation of the best possible hand for each player, and also comparing them to see which one is the winning hand amongst the various players.

- *"As a programmer, I want my network capabilities to update the GUI on each of the players displays, and properly receive information from the players so that the game runs according to how it logically should work."*

  The network was handled with the use of multiple sockets, one for each player, that connects them to a server. The players, or clients then interact with the server by sending them information when they make a move of some kind. The server then processes this and updates the player's view based on the changed information. Most of this code is handled in a broken up controller, one for the server side of the interaction, and another for the client side of the interaction.

Class Descriptions

Class name: Game	
Description: Control and operate the game	
Responsibility	**Collaborators**
+ Create new game + Deal cards to the table and the players + Check each players bet + Find players who have the winning hands + Show cards on Flop, Turn, River, Showdown + Calculate pot money and give to winning players	Player Card Deck Poker Controller Pot GameState

The Game class is the main class of our Model, it is in charge of creating new games, dealing cards, checking the players bets, calculating the winner for each round, and calculating how the money should be divided amongst the players after each round. This class contains objects of the Player class and has its own Deck, from the Deck class. The Game also uses the Pot class to handle the pot for keeping track of how much is currently being bet by the players. The Game also uses the Poker Controller class to updates the model based on the input from the view and changes the view according to the updated model. The Deck class simply contains the 52 Cards, from the Card class. Lastly the Game class also uses the GameState enumeration to determine in which part of the round the players are currently.

Class name: Player	
Description: Represents the player in the game	
Responsibility	**Collaborators**
- Generate best hand - Bet money to the Pot - Flip cards	Hand Card Game CardUtility PlayerState

Every Game consists of 2 to 10 players (according to international poker rule). Every Player object has an integer value that represents the amount of cash they have, an ArrayList of two Cards to represent the cards they are dealt and the best Hand object that stores the best hand. The Player has a PlayerState enumation class to determine whether the player is still playing, make an all-in or folds his dealt cards. If we implement a user account system for future development, we can easily make each Player object into an account by adding username, password encryption and other personalized information

Class name: Hand	
Description: Represents the Hand	
Responsibility	**Collaborators**
- Represents the Hand that player has - Compare itself to other hands	Player Game CardUtility Hand

Every player has a Hand object which represents the best hand they made from 7 available cards. The implementation of the Hand to be quite difficult as each type of hand has different ranks, different attributes and different ways to compare itself to each other. However, we still prefer to implement Comparable<Hand> so that we can sort the hands of every player in order and easily determine the players with the best hand. We finally decided to make Hand a super class and every Hand type a subclass. Each Hand type will an integer value that represents the rank of the hand. The compareTo method in the Hand super-class will compare the rank of the hands, and if they happen to have the same rank, then we will call the compareTo method in each sub-classes to compare the two hands appropriately.

Class name: Pot	
Description: Represents the pot	
Responsibility	**Collaborators**
- get players' cash	Player Game

The Pot Class is a rather small class and is simply responsible for getting the players cash, or how much they bet. To this aim it has interactions with the Player class, in order to get the money, and the Game class.

Class name: Card	
Description: Represents a card with number value and suit	
Responsibility	**Collaborators**
- Generate new card - Compare itself to other cards	Deck Hand Hand sub-classes (High Card, Pair,...) CardUtility CardState Suit

Every player has various Cards that they use along with the games Cards to make the best possible hand that they can. The Card class is responsible for generating cards and being

able to compare itself to other cards and is needed in many other classes. Such as a deck which is made up of 52 Card objects, or the Hand class and its sub-classes who use Cards in order to generate winning hands in poker. The Card needs to have a Suit, which is handled with an enumeration known as Suit. The CardState helps us to determine if a Card is either being displayed, i.e. FACE_UP or FACE_DOWN.

Class name: CardUtility <<EmpUtility>>	
Description: Generate different kinds of Hand and check different kind of Hand	
Responsibility	**Collaborators**
- Generate the best Hand object based on the given 5 cards.	Hand Hand sub-classes (High Card, Pair,...) Card

Finally Cards interact with the CardUtility which is what helps us to determine which of the possible hands that the cards can generate is the best. CardUtility would sort the 5 cards and repeatedly attempt to generate the Hand object from the best type possible (Royal Flush) to the worse. If 5 cards fail to meet specific requirements, the method would return null and CardUtility will try to make a Hand object with a lower rank.

Class name: PokcrController	
Description: Control the model and connect it to the view	
Responsibility	**Collaborators**
- Get and process action from the player - Update view from model	Poker Main Player Main View

The PokerController class is used to get the actions that the client Players are doing so that the model can be updated. In general it is used to connect the model to the views of the various players connected to the server. This required it to work in tandem with three classes. The player in which it takes in data from the player and uses it to update the Poker Main. Also this input data is and updated model is used to update the MainView.