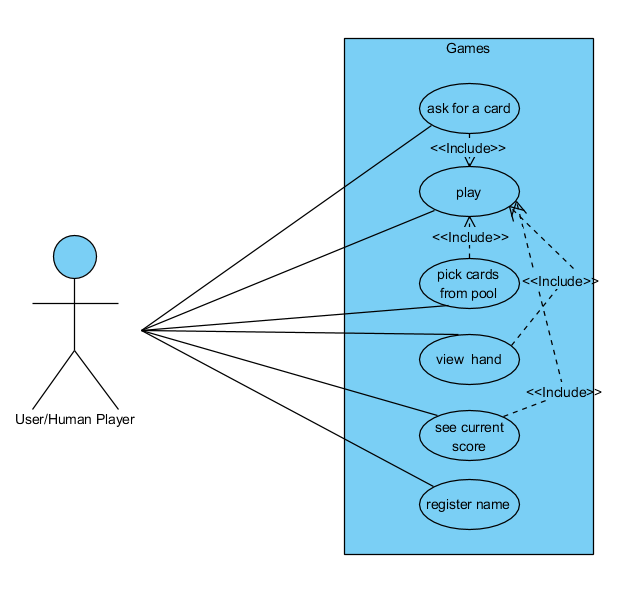
**Group Members**

**Deep Shah**

**Gilbert Napa**

**DELIVERABLE 2**

**Use Case Diagram:**



**Alternate Path:**

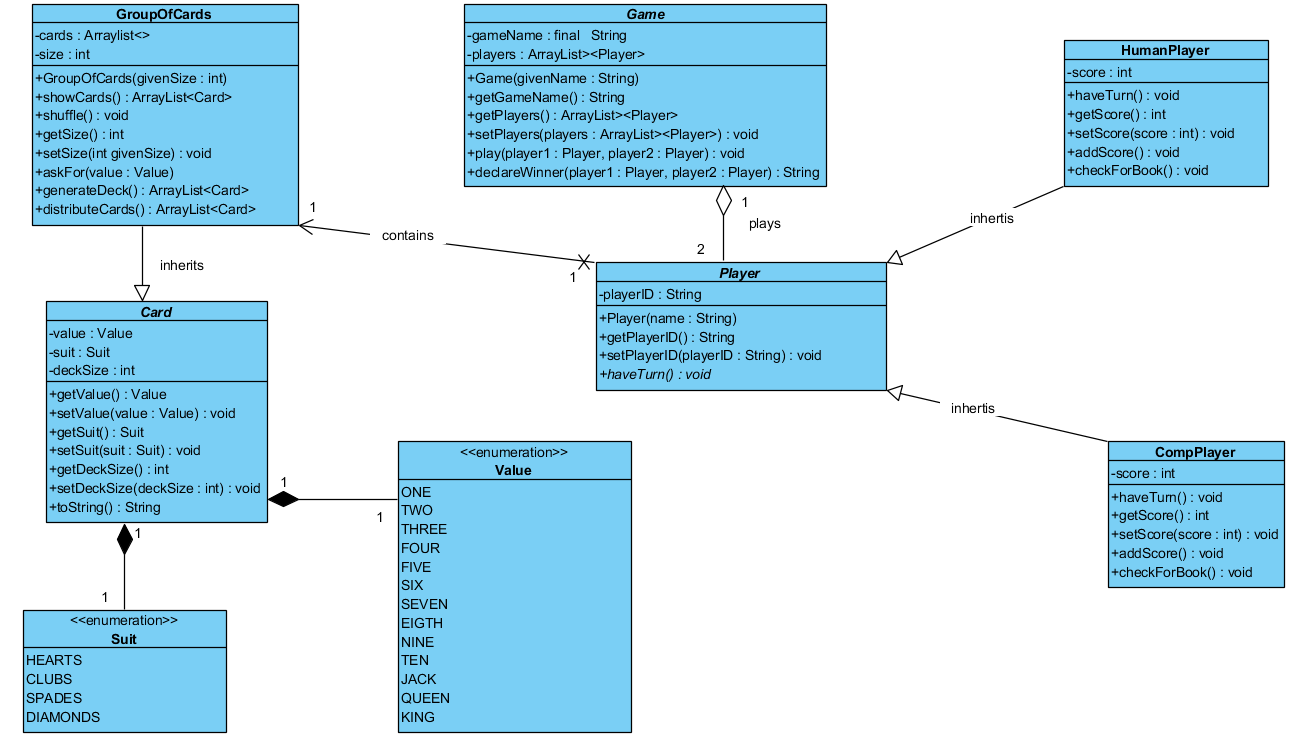
When the haveTurn() method is called for user then it will first ask user whether they want to see their score, see their hand or to ask for card.

Here the play use case does not have an alternate path because it includes ask for a card, pick cards and other use cases for which alternate path is defined below.

See current score does not have an alternate path because it only shows the score of the user there is no user interaction.

1. Ask for a card:
   1. Enter the value of a card to ask to computer
   2. User enters some wrong value when asking for a card.
   3. Prints an error message indicating to that wrong value was entered
   4. Prompt again to enter value of a card to ask.
2. Pick cards from pool
   1. Computer does not have the card the user asked for.
   2. User picks a card from the card but there are no cards left in the pool.
   3. Prints a message that there are no more cards in the pool
   4. Game continues
3. View Hand
   1. User selects to view their hand
   2. There are no cards in their hand
   3. Prints a message that there are no cards in their card
   4. Game continues
4. Register Name:
   1. Enter name
   2. User enters numbers or anything except letters
   3. Prints an error message to enter name with letters
   4. Prompts to enter name again.

**UML Class Diagrams:**



**DESIGN DOCUMENT TEMPLATE**

**Project Background and Description:**

After the feedback from deliverable one our team has decided to change some of the rules of the game.

**Rules before:**

* Can be played by 3-4 players.
* The dealer (dealer is selected randomly) distributes 5 cards among the players and the rest of the cards are kept as a pool (remaining cards).
* First turn goes to the player next to the dealer.
* Player A can only ask for a card to player B if player A has the requested card in hand.
* If the player A asks for a card to player B but does not have that card, then player B says, “Go Fish” which tells the player A to draw a card. Then after, the turn then goes to player B. If player B has the requested card, player B gives all the cards of the rank that is requested.
* If the player runs out of card, player picks up five cards from the pool.
* If there are no cards left in the pool, then the player is out of the game.
* Once a player gathers all the cards of a book (four of a kind), the player must place it down and reveal the cards in front of other players; that players earns a point.
* The player who gathers the most books when the card pool is finished wins the game5

**Rules now:**

* The game will be played by two players. First one will be the user/ human player and the second player will be the computer.
* Before the players begin the game, 10 cards will be distributed evenly to both players. (We decided to increase the initial number of cards because we decreased the number of players)
* First turn goes to the human player.
* User can only ask for a card to computer if the user has the requested card in hand and vice versa.
* If the user asks for a card to computer but the computer does not have that card, then computer says, “Go Fish” which tells the user to draw a card. Then after, the turn then goes to computer. If computer has the requested card, computer gives all of the cards of the rank that is requested.
* If the player runs out of card, player picks up five cards from the pool.
* Once all the books are all collected then the game declares the winner with the highest number of books.

Our game will run until all the books are collected. Player would be able to look at his/her score and the hand before his/her turn. Also, before the beginning of the game the player would be able to register his/her name.

The game would have the ability to prompt the user to enter correct details if any exception occurs. These exceptions just not include java errors but also the game errors, for instance the user or computer asks for a card that they do not have in their hand. The game would be handling all such exceptions.

**Design Considerations:**

**UML description:**

As shown in the UML, We have a “Card” class that has two Enum classes named “Suit” and “Value”. Then Enum “Suit” has a value of Hearts, Clubs, Spades and Diamonds while the Enum “Value” has a value of One, Two, Three, …, Ten, Jack, Queen, King. This then explains that, for one Card, there is only one Suit and one Value.

Also, we have a “GroupOfCards” class which inherits the “Card” class. This class contains two data fields: cards of an arraylist and size of an int type. This class contains methods that generates the deck in hand and distributes the cards. It also has a method which ask for a card to the other player.

For the “Player” class, it has two subclasses: “HumanPlayer” and “ComputerPlayer”. The “Player” class is an abstract class with an abstract method called haveTurn(). Human player and computer player classes overrides this abstract method and implements it in their own way.

The “Game” class contains two data fields: gamaName which is a constant String and players which is an arraylist of the “Player” class. “Game” and “Player” class has an aggregation relationship where one game can be played by two players.

1. **Encapsulation**: All our classes will follow these rules, all the data fields in each class will be private, and setters and getters would be implemented for them.
2. **Delegation:** The game class delegates the player class as in the game class player object is used as instance variable. Similarly in Card class instance variables are used of enumeration classes Suit and Value. This also helps to maintain high cohesion.
3. **Cohesion**: All our classes follow high cohesion as they do what they are supposed to do. For example, the game class only has methods which relate to the game like getGameName(), setGameName() and so on. Similarly, Player class only handles the method related to player, like method for player’s name, method for their score and so on.
4. **Coupling**: The card class and enumeration classes have low coupling as any changes made in enumeration classes will not affect the card class. All the methods in card class will run without any errors. Similarly, all other classes follow low coupling, where one class won’t affect any other class if any changes are made.
5. **Inheritance**: The human player and computer player class inherits the player class as both the player have something in common like name.
6. **Aggregation**:  Player class and game class has an aggregation relationship as the player class would exist or would make sense without the game class.
7. **Composition:** Here the Card class has composition relationship with Suit class (which is enumeration) and the Value class (enumeration class) because a card cannot exist without a suit or a value.

8)     **Flexibility/Maintainability:** Our goal is to write a code in such a way that if any feature or rule of the game is changed in the future, then it will be easy to change that code so that it does not affect any other code. For instance, if the number of suits or value changes then the changes would be made only in their respective enumeration classes plus this would not affect any other part of the codes.