I. Unified Modeling Language Design

Project Description

The project is about designing a simple console-based card gambling system for a local casino using C++. The development is divided into four sections, each building on the previous one.

- ### Section A

  Objective: Develop a program to display cards.
  Functionality: The program takes two inputs from the user: the card number and the card type. It then displays the corresponding card on the console.

- ### Section B

  Objective: Expand the system to accommodate multiple players.
  Functionality: The program should handle between 1 to 4 players, including the computer (NPC). Each player receives 3 cards, and the program tracks which player gets which cards. The cards for human players are displayed, while the computer's cards are hidden.

- ### Section C

  Objective: Implement poker hand combinations to determine a winner.
  Functionality: Using the cards dealt to each player and the computer, the program should determine the winning hand based on standard poker rules. It compares each player's hand against the computer's to identify the winner.

- ### Section D

  Objective: Develop or implement an additional card game.
  Functionality: Using the code developed in previous sections, we create another card game thereby simulating another game instance as the cards are randomly selected and assigned to players.

Project Actors

- **Player**: A person interacting with the card game system, having a hand of playing cards.
- **System (NPC)**: The computer or console system which deals cards,manages the game and is also a Player though we don't see its hand of cards..

Use Case Diagram

The use case diagram includes the following primary use cases:

- **NPC**:
  - Display Cards: Displays cards based on user input.
  - Deal Cards: Deals (Share) to Players and itself.
  - Track Cards: keeps track of which player has which cards.
  - Display Player Cards: System displays the cards for non NPC players.
  - Determine Winner: System evaluates the poker hands to determine the winner.
  - Play Additional Game: System handles an additional card game with new entries..

Class Diagram

The class diagram includes the following classes with relationships:

Classes

- Card
  - Brief: Encapsulates the properties and behaviors of a card.
  - Attribute(s): number, suit
  - Method(s): getCardNumber(), drawCardLayout(), getCardTypeSymbol(). getNumber(), getSuit()

- Player
  - Brief: Represents a participant in the game with a hand of cards.
  - Attribute(s): hand
  - Method(s): receiveCard(Card), getHand()
- Hand
  - Brief: Represents the cards held by a player.
  - Attribute(s): cards
  - Private method(s): isFlush(), isStraight(), isThreeOfAKind(), isPair(), isMiniRoyal()
  - Public method(s): addCard(), getCards(), getHandRanking(), getHandRankingValue(), handRankingToString(), compareHands()
- NPC (inherits from Player)
  - Brief: Includes additional behavior specific to the computer player. Mainly dealer and game manager behaviors.
  - Attribute(s): N/A
  - Method(s): shuffleDeck(), dealCard()
- Deck
  - Contains a collection of Card objects and methods to shuffle and draw cards.
  - Attributes: cards
  - Methods: shuffle(), drawCard()

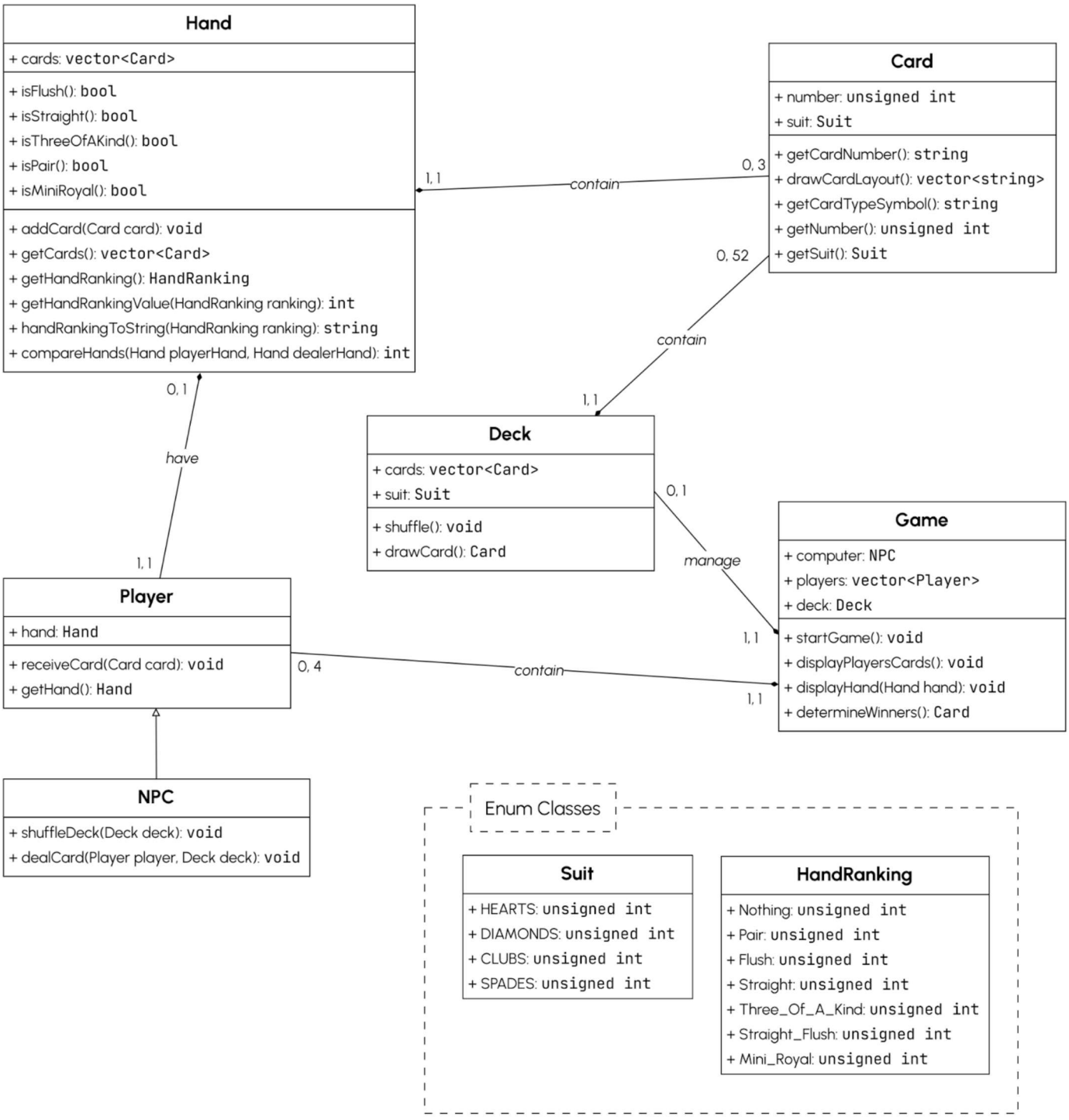
- Game
    - Attributes: computer/NPC, players, deck
    - Methods: startGame(), displayPlayersCards(), displayHand(), void determineWinners()

Relationships

- **Game Contains Players and NPC:**
    - **Game** *has* multiple **Player** objects and one **NPC** object.
    - The **NPC** *inherits* from **Player**.

  Game is responsible for managing the game flow, including dealing cards and determining winners.
- **Game Manages Deck:**
    - **Game** *has* a **Deck** object which is used to deal cards to the players and the dealer.
- **Player and NPC Have Hands:**
    - Each **Player** and the **NPC** (since NPC inherits from Player) *has* a **Hand** object.
    - The **Hand** object *contains* **Card** objects that represent the cards in the player's hand.
- **Deck Contains Cards:**
    - The **Deck** object *contains* multiple **Card** objects. It provides methods to shuffle the deck and draw cards from it.
- **Hand Contains Cards:**
    - The **Hand** object *contains* multiple **Card** objects and provides methods to add cards and determine hand rankings.

Diagrammatic representation



II. Object Oriented Implementation

For the implementation of this project, we will be using the Visual Studio (Community) IDE and the C++ programming language.
We will go through each section of the project and give details on the class implementations and at last the main() program implementation.

File Structuring:
It is important to note that the project follows the default file organization of Visual Studio which consists of putting together the header files (.h) in a Header Files folder and source files (.cpp) in a Source Files folder.

1. Section A

In this section, we mainly focus on creating a **Card** class and displaying it to the console with a convenient and user-friendly layout.

Class Card

Attribute(s):

- `unsigned int number`: serves as the card number and ranges from 1 to 13.
- `Suit`[1] `suit`: serves as the card type and belongs to the enum class Suit which consists of `SPADES, CLUBS, HEART and DIAMONDS`.

Method(s):

- `Card(unsigned int number, Suit suit);`
  Constructor. Initializes the card with a number and type.
- `unsigned int getNumber() const; Suit getSuit() const;`
  Getters for the Card number and suit respectively for external access.
- `std::string getCardNumber() const;`
  Returns the string representation of the card number (e.g., "A" for Ace, "J" for Jack, etc.).
- `std::string getCardTypeSymbol() const;`
  Returns the string representation of the card suit (e.g. ♥ for HEARTS, ♦ for DIAMONDS, etc).
- `std::vector<std::string> drawCardLayout() const;`
  Constructs and returns a vector of strings representing the card layout.

Enum Class Suit

- Variables: `HEARTS, DIAMONDS, CLUBS, SPADES`

[1] In playing cards, a suit is one of the categories into which the cards of a deck are divided. The suitmarks of the international, or standard, deck indicate two black and two red suits—namely **spades, clubs, hearts**, and **diamonds**.

Extra

- **Formatting with the `{fmt}` lib**: For the proper display of the cards layout, some special characters were used namely, **box drawing characters**[2]. This required the addition of an external open source library known as fmt (https://github.com/fmtlib/fmt) to display these characters and launching the console with the **/utf-8** flag. The cost of a user-friendly console application.
- **Displaying cards side by side**: Here was another challenge in this section. So, we came up with a utility function, `void displayCardsSideBySide(const std::vector<Card>& cards);` that does just that. The idea is, in the Card class we draw the Card layout and store it in a vector and only print it out to the console when this utility function is called so as to handle the side by side positioning of cards to simulate a hand.

2. Section B

We keep in mind the above implementation of the Card class and will be using that to implement the following:

Class Deck

Attribute(s):

- `std::vector<Card> cards;` represents a list/container of playing cards during a game session

Method(s):

- `Deck()`: Constructor.The Deck constructor directly initializes a Deck with a set of 52 playing cards of all suits and numbers.
- `void Deck::shuffle();`
  This method shuffles the deck of cards using the Fisher-Yates[3] algorithm, an efficient and unbiased way to randomly order a list.
- `Card Deck::drawCard()`
  This method draws a card from the deck and returns it. This method removes the last card from the deck and returns it.

Class Hand

Attributes:

- `std::vector<Card> cards;` represents a list/container of cards possessed by a Player.

Methods:

[2] Sources:
- http://xahlee.info/comp/unicode_drawing_shapes.html
- https://coolsymbol.com/corner-symbols.html

[3] Fisher-Yates algorithm source: https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

- `void addCard(const Card& card);`
  Adds a Card object to a Player's Hand
- `const std::vector<Card>& getCards();`
  Getter function that returns a Player's Hand cards.

Class Player

Attributes:

- `Hand hand;` contains the cards assigned to a Player after shuffling by the NPC.

Methods:

- `Hand& getHand();`
  Getter function that returns the Player's hand. Notice that we get the Hand and not the cards. Getting the cards is implemented in the Hand class.
- `void receiveCard(const Card& card);`
  Adds a card to a Player's Hand.

Class NPC

This class has a public inheritance on Player as the NPC is also a Player in the system but has other roles such as dealing the cards and managing the Game.

Attributes:

- No specific attributes.

Methods:

- `void shuffleDeck(Deck& deck);`
  This method shuffles the deck of cards.
- `void dealCard(Player& player, Deck& deck);`
  This method deals a card to a player.

Class Game

Attributes:

- `NPC computer;`
- `std::vector<Player> players;`
- `Deck deck;`

Because a Poker Game basically consists of a Dealer (in this case the NPC), Players and a Deck of playing cards.

Methods:

- `Game();`
  We directly initialize the constructor here with the number of players we want in a Game.
- `void startGame();`
  This method starts the game by shuffling the deck, dealing cards to the players, and dealing cards to the computer.

- `void displayHand(const Hand& hand);`
  This method displays the cards of a hand side by side.
- `void displayPlayersCards();`
  This method displays the cards of each player and the computer.

Extra

- It is important to note that the rest (if not all) of the Classes needed to be implemented already in this section to be able to simulate the sharing and displaying of Cards.
- **Efficient card shuffling**: With the current implementation, it is not possible for two players to have the same cards. Here is why:The Deck class is designed to manage a standard deck of 52 cards.When the `Deck::drawCard()` method is called, it removes the card from the cards vector.This ensures that once a card is drawn, it is no longer available to be drawn again.
- **Runtime persistence of each Player's cards**: The Deck class is designed to manage a standard deck of 52 cards.When the `Deck::drawCard()` method is called, it removes the card from the cards vector.This ensures that once a card is drawn, it is no longer available to be drawn again.
  Also, the fact that the players' hands are persistent during runtime, allows us to keep track of which cards belong to whom.

3. Section C

In this section we focus on extending the functionalities (methods) of some classes developed in the previous section, eventually the Hand and Game classes and adding a HandRanking enum class that will define the possible hand rankings (e.g. Straight flush, Three of a kind, Mini royal, etc)
The aim of these new functionalities is to have a working game that takes into consideration all the combinations of a 3 cards poker game[4].

Enum Class HandRanking

Variables: `Straight_Flush, Three_Of_A_Kind, Straight, Flush, Pair, Mini_Royal, Nothing`
This enum class represents the various possible hand rankings we can obtain during a game.

Class Hand (Improved)

New method(s):

- `bool isFlush() const;` `(private)`

[4] 3 cards poker game rules:: https://www.table-games-online.com/3-card-poker/hand_ranking.html

This method checks if the hand is a flush. It returns True if the hand is a flush, false otherwise.
Here we only want to check if the cards are all of the same suit.

- `bool isStraight() const;` `(private)`
  This method checks if the hand is a straight. True if the hand is a straight, false otherwise.
  Here we check if the cards are in a sequence. For a best case scenario, we sort the cards in ascending order before comparing them and the difference in card number should be 1 across all 3 cards.
- `bool isThreeOfAKind() const;` `(private)`
  This method checks if the hand is a three of a kind. True if the hand is a three of a kind, false otherwise.
  Here we check if the cards are all of the same rank or number independent of the suit.
- `bool isPair() const;` `(private)`
  This method checks if the hand is a pair. True if the hand is a pair, false otherwise.
  To achieve the pair combination, we create a map of key-value pairs, eventually `key-count_of_card_number` pairs and at the end we get the size of the map. If this map size is exactly 2, then we have a pair as at most 2 cards are of the same number and the third one is different.
- `isStraightFlush (isFlush() && isStraight())` `(private)`
  This function is not implemented as it's just a combination of Flush and Straight checks. So, if they both return true, then we have a straight flush, false otherwise.
- `bool isMiniRoyal() const;` `(private)`
  This method checks if the hand is a mini royal. True if the hand is a mini royal, false otherwise.
  The highest straight flush: `A-K-Q` all of the same suit.
- `static int compareHands(const Hand& playerHand, const Hand& dealerHand);`
  This method compares two hands and determines the winner.
  To achieve this, we assigned weights/points to HandRankings with Nothing being the lowest (0) and Mini Royal being the highest (6). In that way, when a player gets a given combination, we can value that combination and compare with that of the NPC/Dealer and make a decision.
  But what happens in case of a tie? Yes it can happen. No worry, in this case, we just sort the cards in ascending order and see who has the highest card number in his hand or you can still see it as summing up all card numbers for each Player, comparing and determining a winner.

Class Game (Improved)

New method(s):

- `void determineWinners();`
  This method gets a specific result from comparing hands and displaying the results.

Utilities

For the proper working of these classes and for a more enhanced user experience, we added some utility methods to the Hand class including:

- `HandRanking getHandRanking() const;`
- `static int getHandRankingValue(HandRanking ranking);`
- `static std::string handRankingToString(HandRanking ranking);`

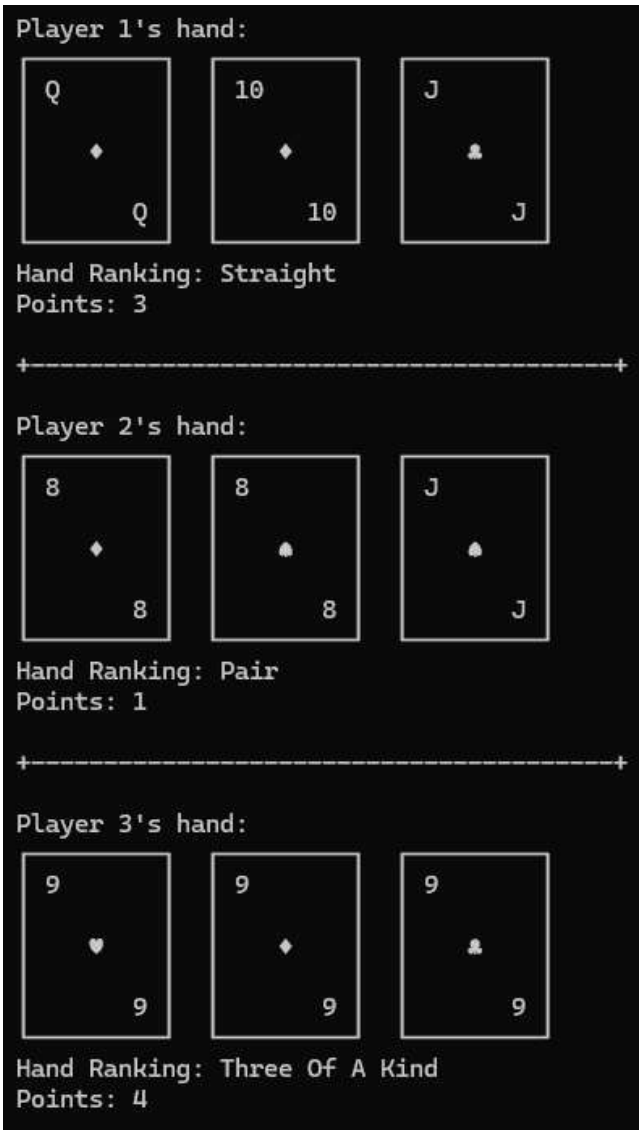
All of these basically act as getters and return data in a more readable and useful format.

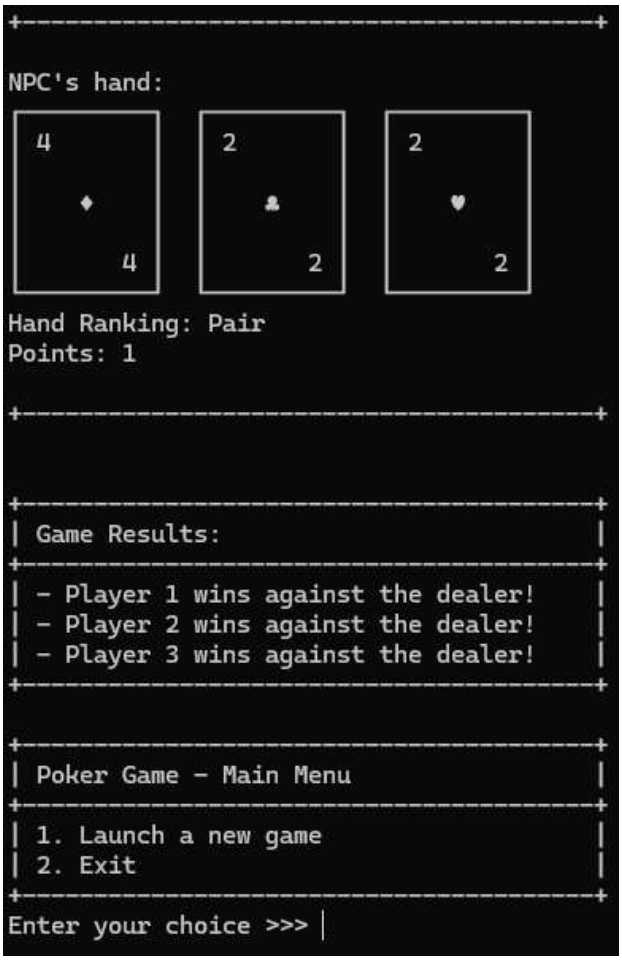
4. Section D

At the end of this project, we obtain a console application with a considerable amount of emphasis on the user-friendly part of it.
To play, it's pretty simple:

1. Run the Poker Game from the Poker_Game.cpp file which contains the `main()` function.
2. From the menu that displays, you can either launch a new game or exit the program. Enter your choice and see the program coming to life.

Output





Extra

The project's source code has been made publicly available on Github: https://github.com/Joel-Fah/Poker_Game