IS442
Object Oriented Programming
Group Project Report

The Hearts Game

Prepared for:
Mr. Lee Yeow Leong

Prepared by:
Beh Wei Chen, Teo Jia Cheng, Wong Xin Wei

Content Page

1. Introduction

Hearts is a great trick-taking family card game for kids and adults alike, which involves 4 players.

Game Play

There will be 4 players - the user controlling one hand and the computer controlling the other three hand. The game starts by shuffling the deck and distributing the cards to each player, each player is dealt with 13 cards. Each player will then select three cards and pass them to the player on their left. The passing rotation will be: round 1 - to the player on the left, round 2 - to the player on the right, round 3 - to the player across the table and round 4 - no passing.

The player who has the 2 of clubs will start the game by playing that card in the set. Each player must follow suit, if a player does not have a card of the leading suit, a card of any other suit can be played excluding point card. In the first round, no player can play a card that is a point card. Second round onwards, if a player lacks a card of the leading suit, a card of any other suit can be discarded, if a player discarded a heart card, which means that heart is broken and any player may lead with a heart card. The highest card of the suit wins the set and the winner of the set will leads next by playing the first card. The round continues until all 13 sets have been played.

Each heart card has 1 point whereas the queen of spade has 13 points. However, if one player has won all 13 hearts and the queen of spades, this is termed “shooting the moon”. The player who shoots the moon will get 0 points, whereas the other player will get 26 points. The game will end when one player gets more than 100 points and the player with the least points will win the game.

2. Analysis and design

2.1. Class Diagram

Design class diagram for your application

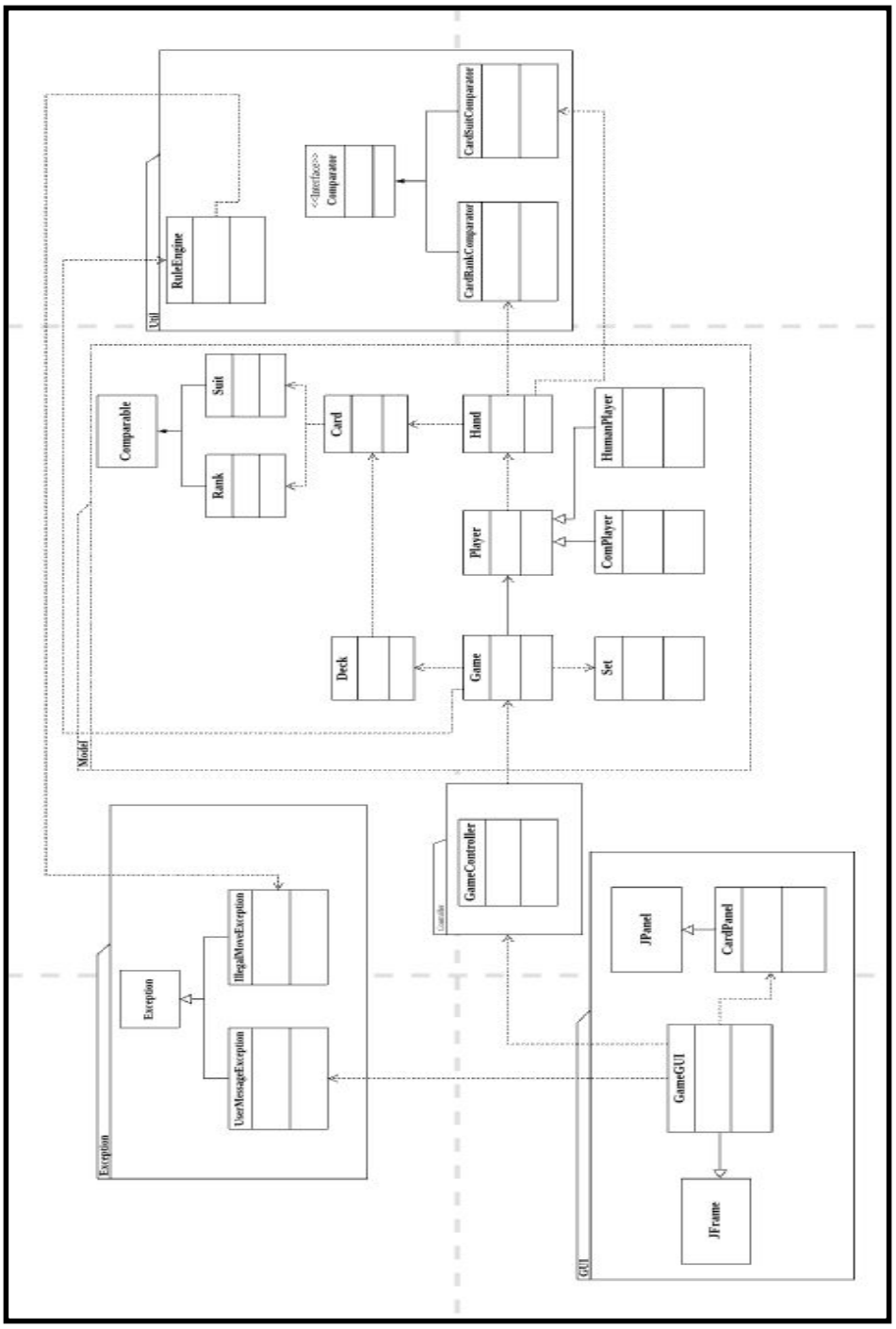


Figure 1: Simplified Class Diagram

The class diagrams has been constructed to match the UML conventions as much as possible.

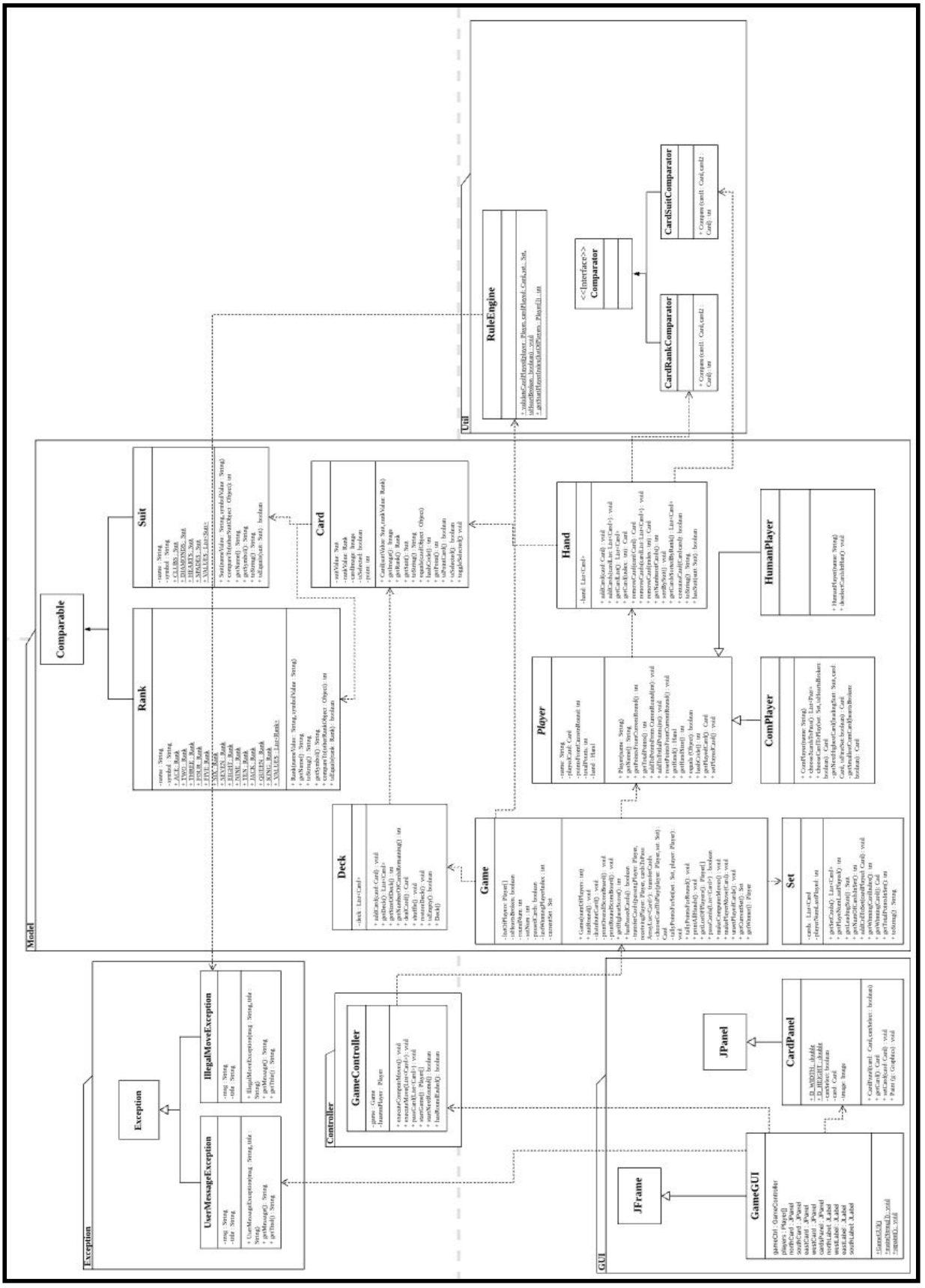


Figure 2: Detailed diagram

Class diagram - Zoomed in

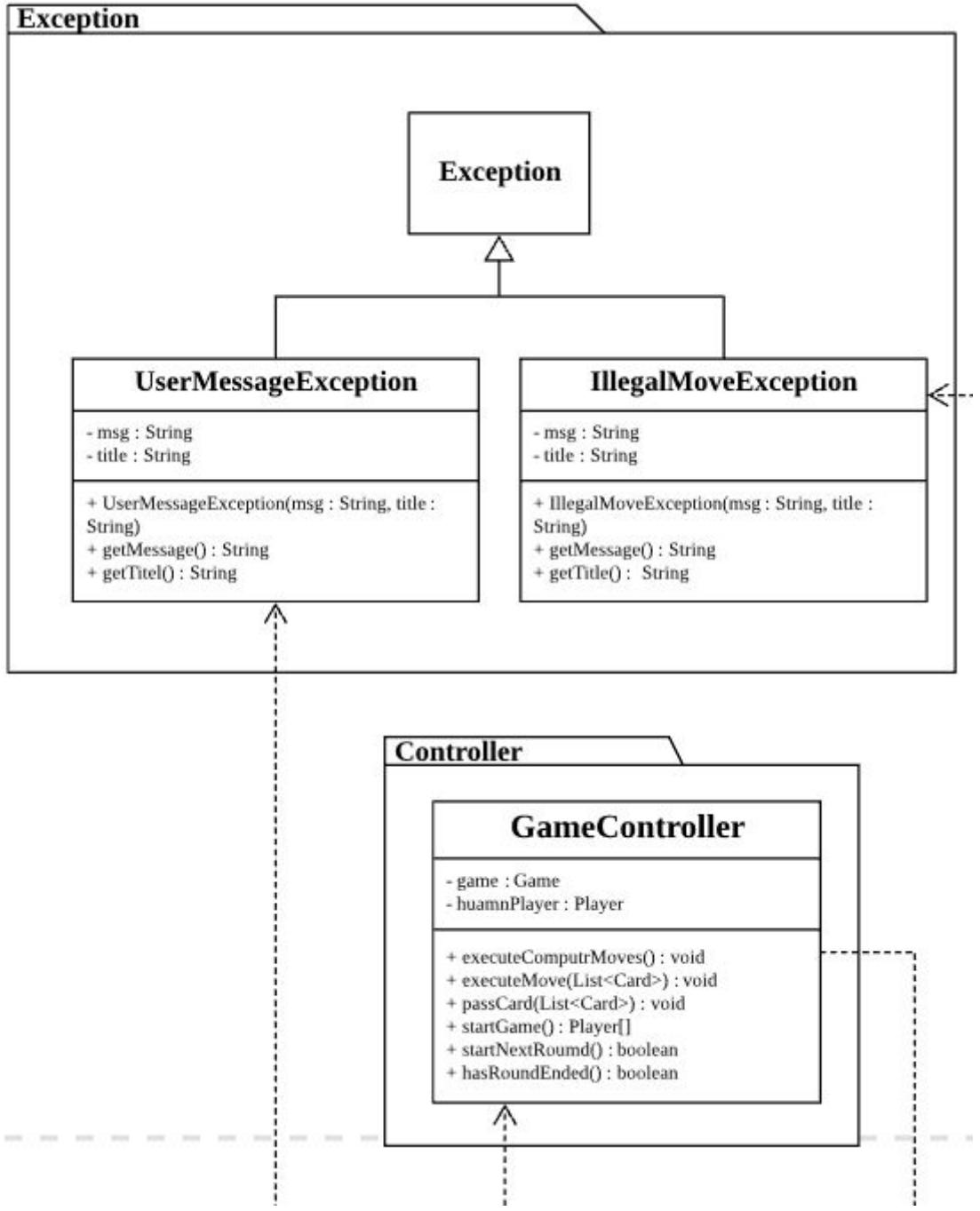


Figure 2.1: Exception and Controller package

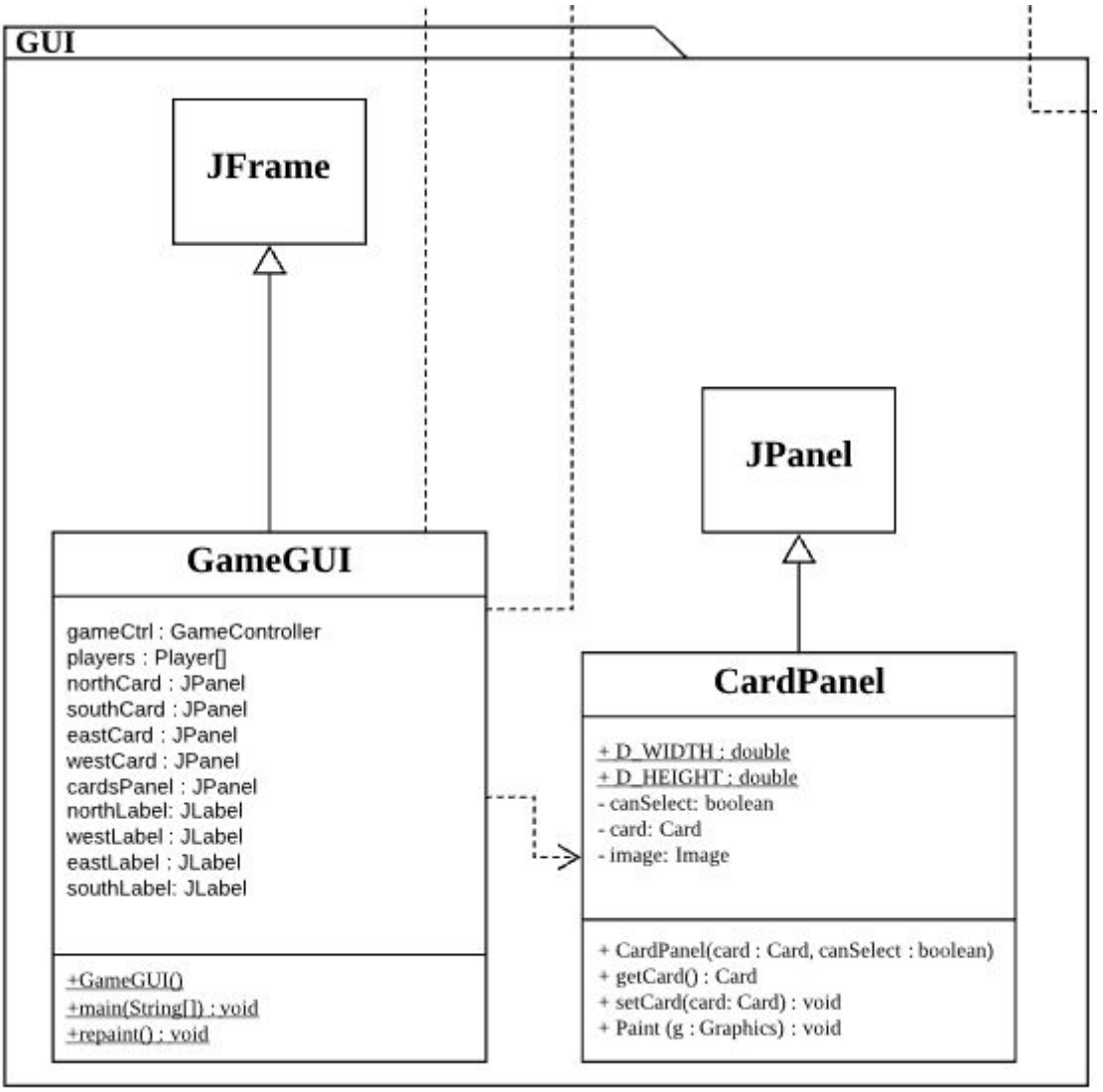


Figure 2.2: GUI package

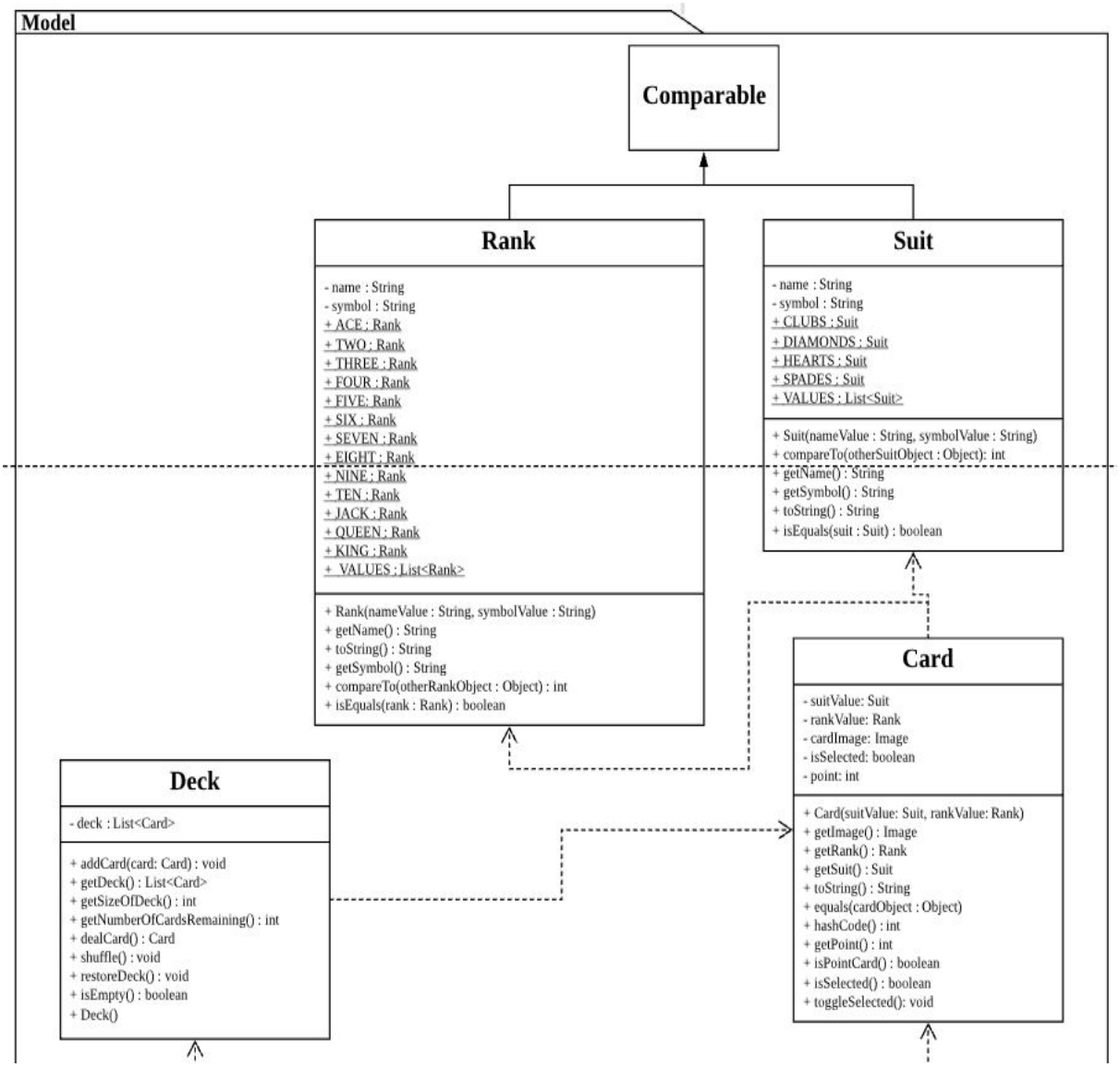


Figure 2.3: Model package top

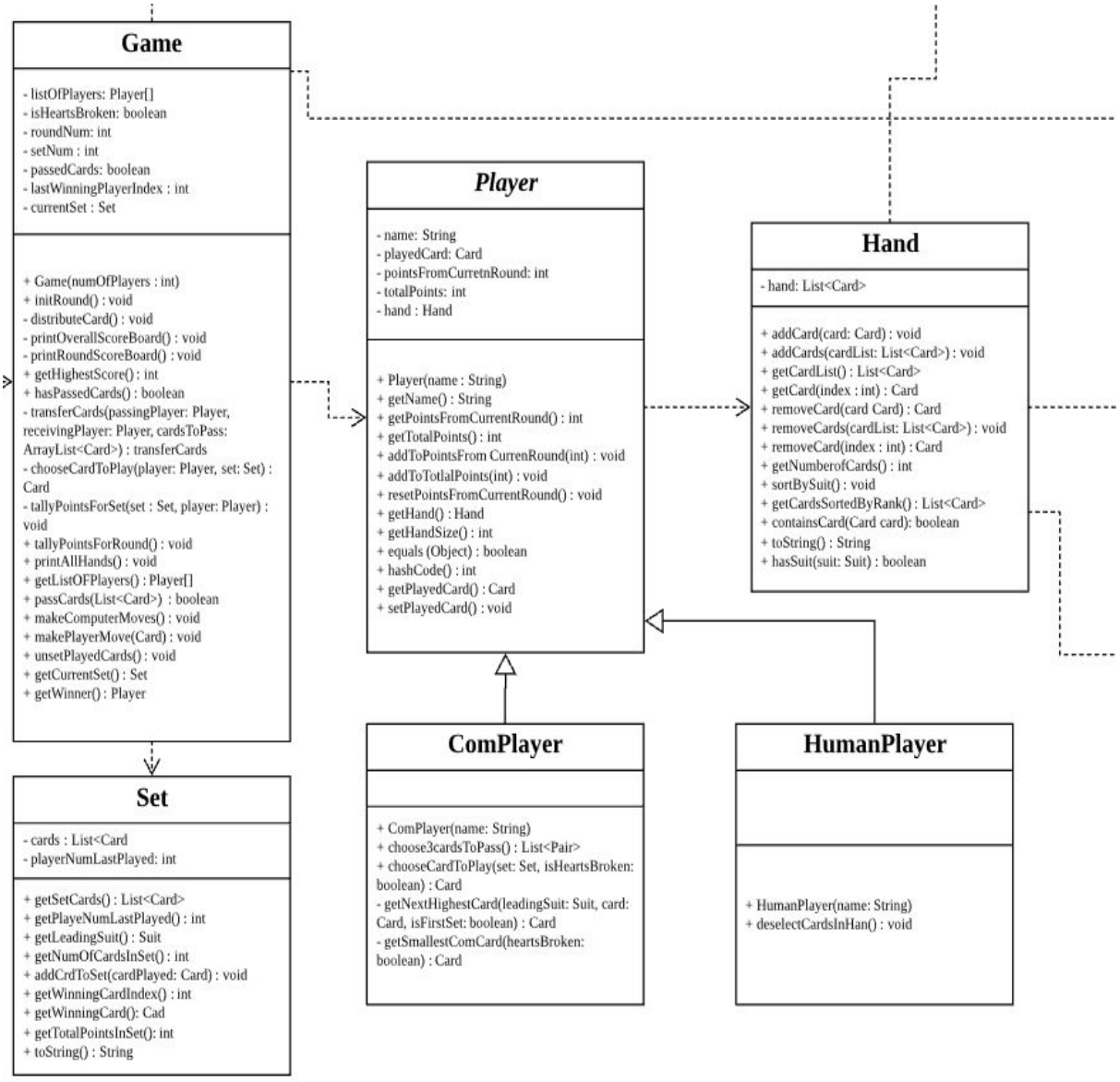


Figure 2.4: Model package bottom

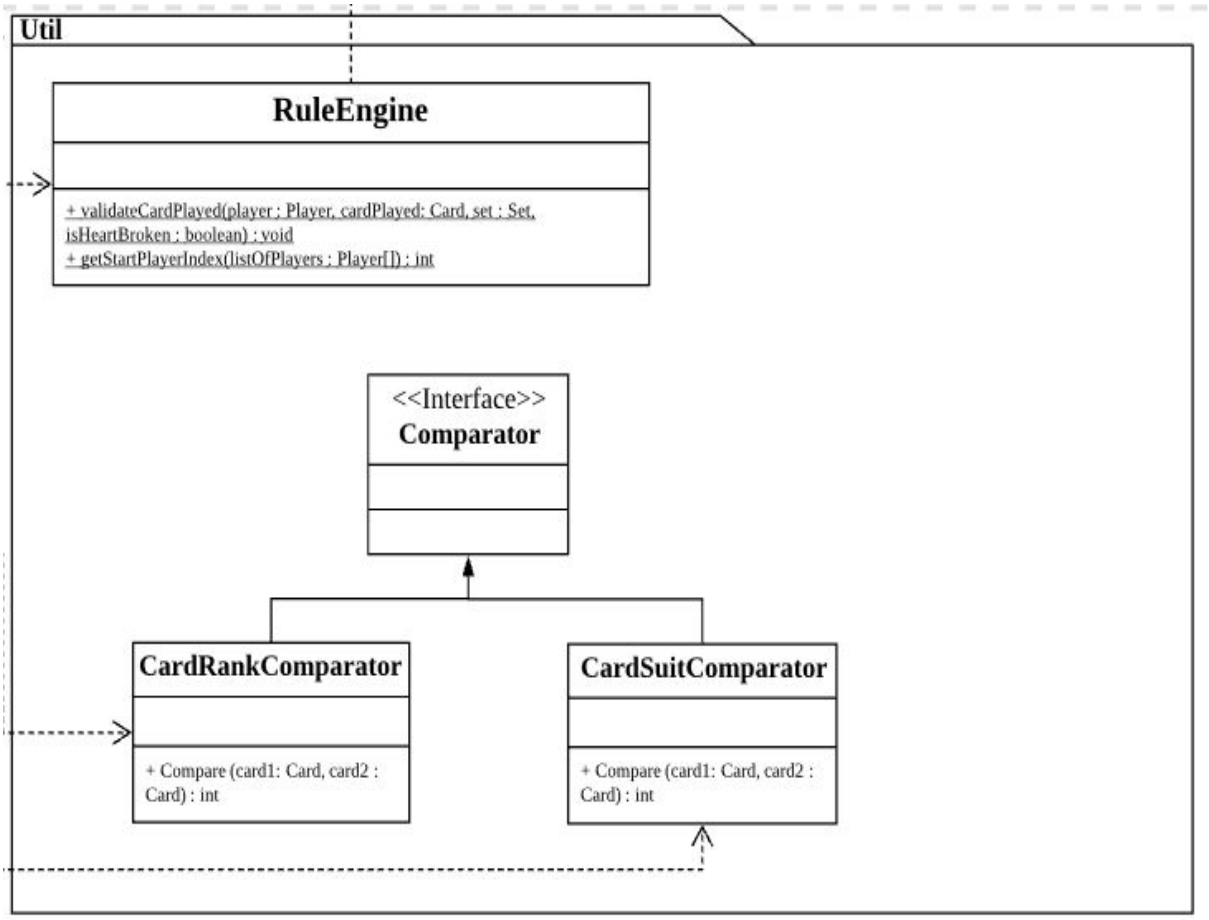


Figure 2.5: Package package bottom

2.2. Object Oriented Design Considerations

We have applied various Object-oriented concepts, such as encapsulation, abstraction, inheritance and exception handling in our code design within our Hearts programme implementation.

Our team took guidance from the builder pattern approach when designing our class diagram. As a builder pattern builds a complex object using simple objects and using a step by step approach. The games of hearts consists of step by step approach where a object uses another object in a sequential manner. For example: GameController uses Game and Game uses Player and Player uses Hand in a linear manner.

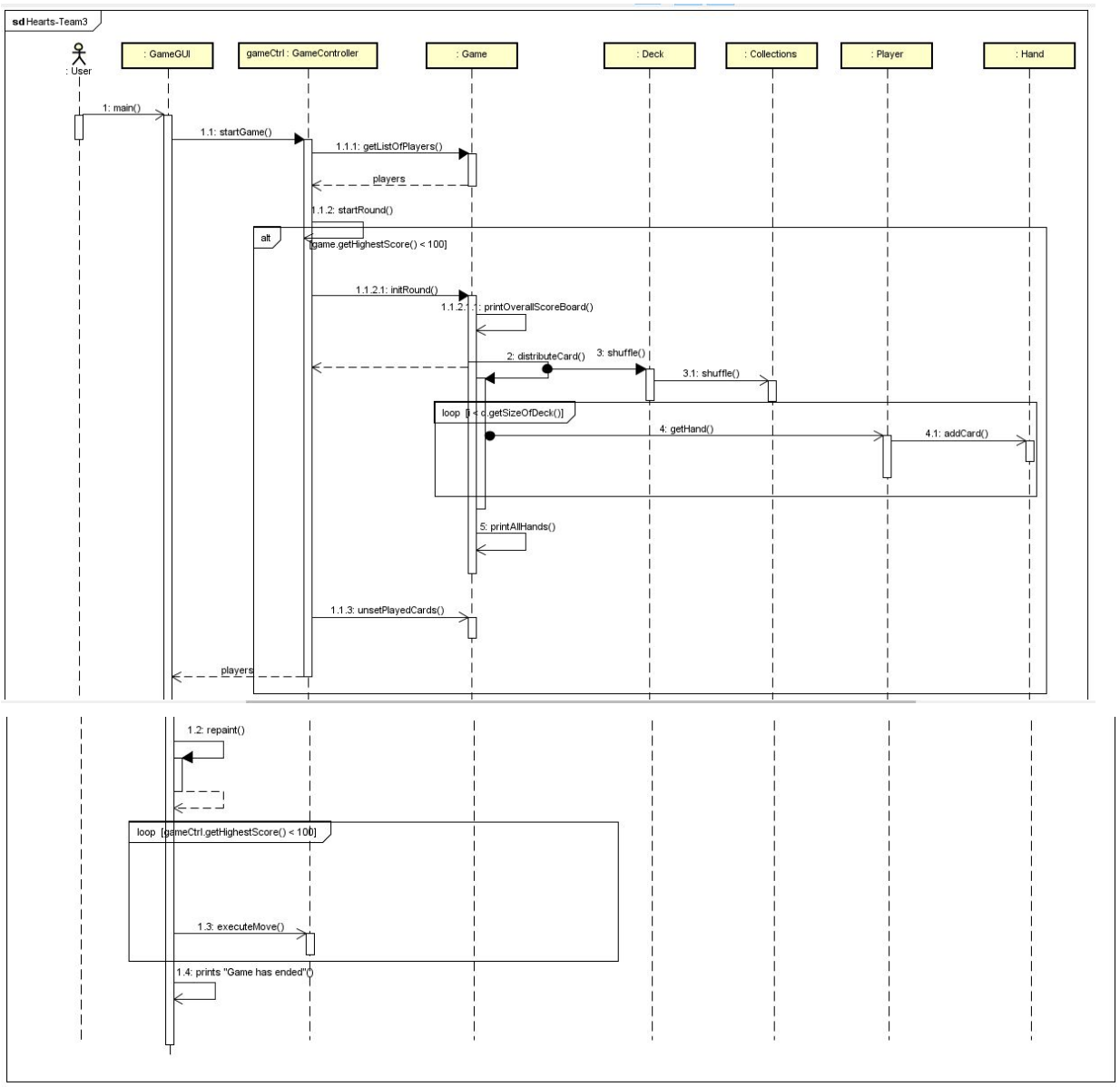
Our classes made extensive use of encapsulation with accessors being used only if deemed necessary. Methods are declared private whenever possible to avoid excessive access to certain class methods by the user. For instance, in the Game class, methods called by other classes were declared as public whereas those that are only called within the class were declared as private. This prevents misuse and confusion between methods between different classes.

We also incorporated inheritance within our classes to avoid having duplicated methods with similar function. In the Hearts game scenario, we have a Player class with all general player functions applicable to the HumanPlayer and the ComPlayer, such as the getHand() and getName() methods. As such, HumanPlayer and ComPlayer are both child classes of Player with additional class methods for functionality. This provides us with the ability to group several related classes together to create a concise structure with no repeated methods. In this scenario, Player was not implemented as an interface class as concrete methods were required as opposed to abstract methods.

In addition, we also implemented the interface class, Comparator, to create CardSuitComparator and CardRankComparator classes which serve to compare Card objects based on both their Suit and Value. This was selected over the interface class, Comparable, as it gave us the leverage to sort the card by both its Suit and Value in different orders. For instance, CardSuitComparator compares Cards by their Suit followed by Rank while CardRankComparator compares Cards by their Rank followed by Suit.

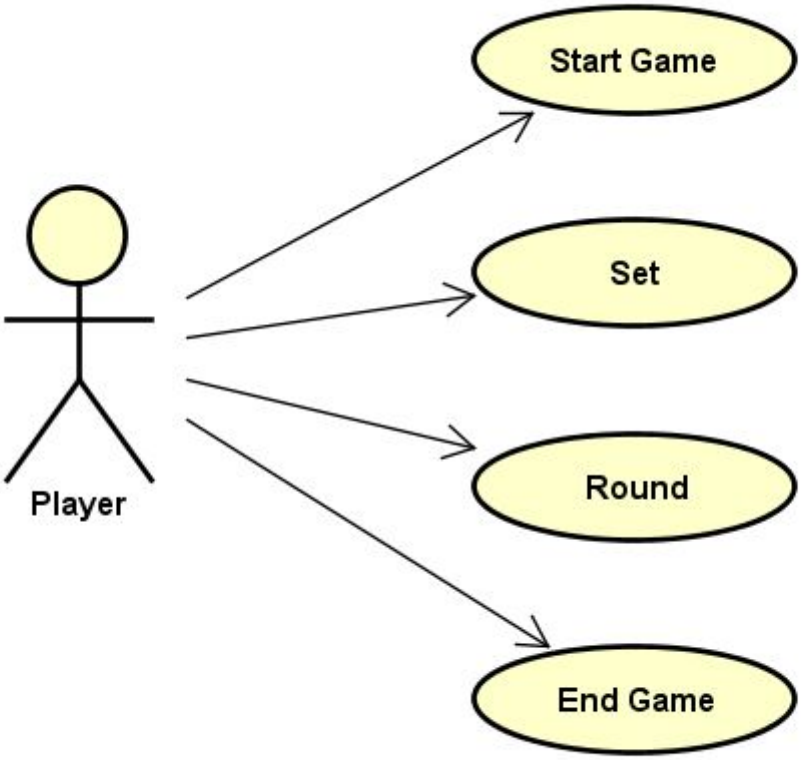
Overall, these are the key consideration we had throughout the course of our project implementation. We found that the object-oriented concepts that we integrated into our project assisted in the usability and readability of our code.

2.3. Sequence Diagram



Specific low-level methods are left out for brevity.

3. Case Diagram

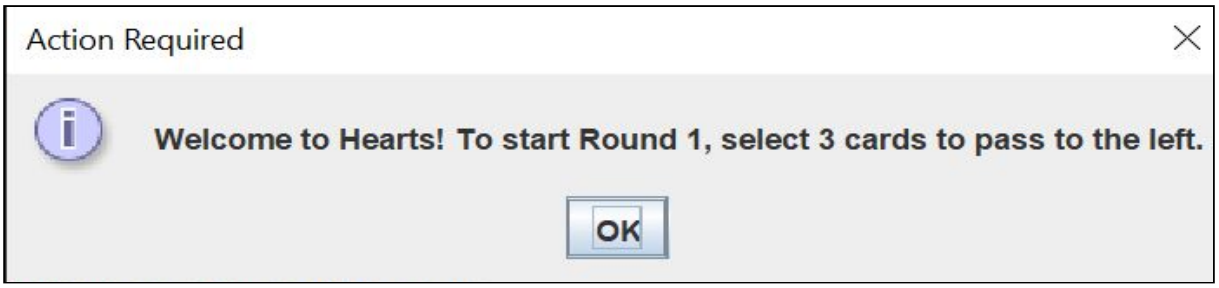


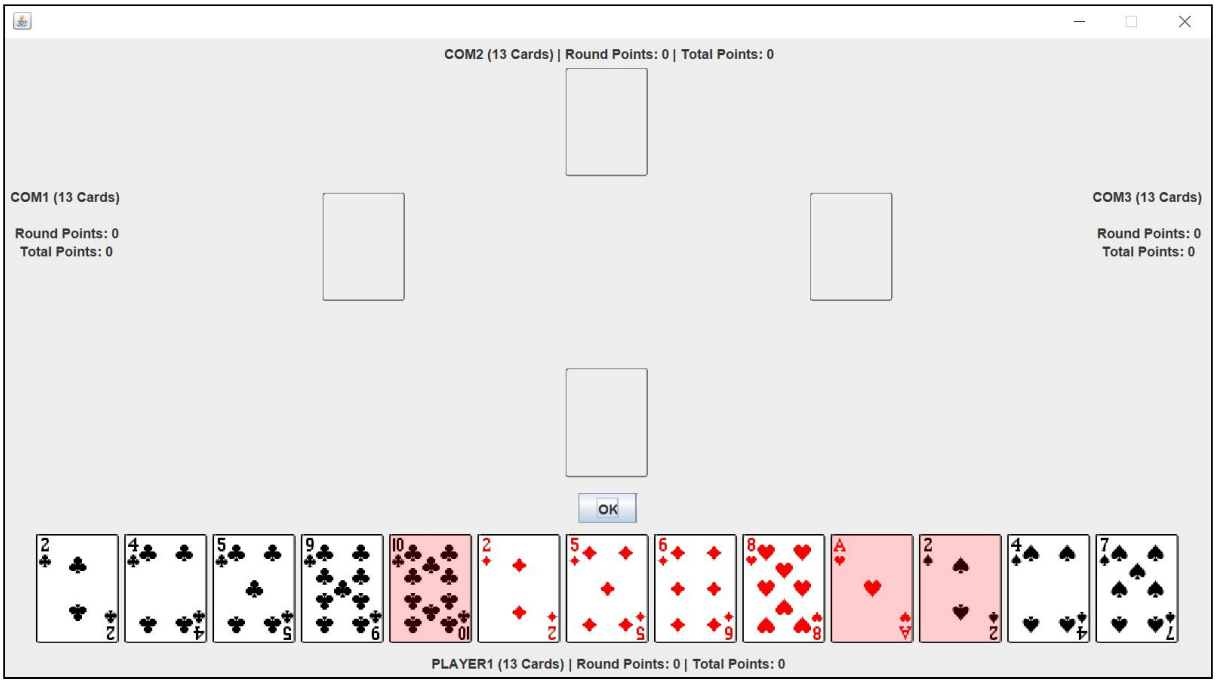
3.1. Use Case Scenario

3.1.1. Scenario 1: Start Game

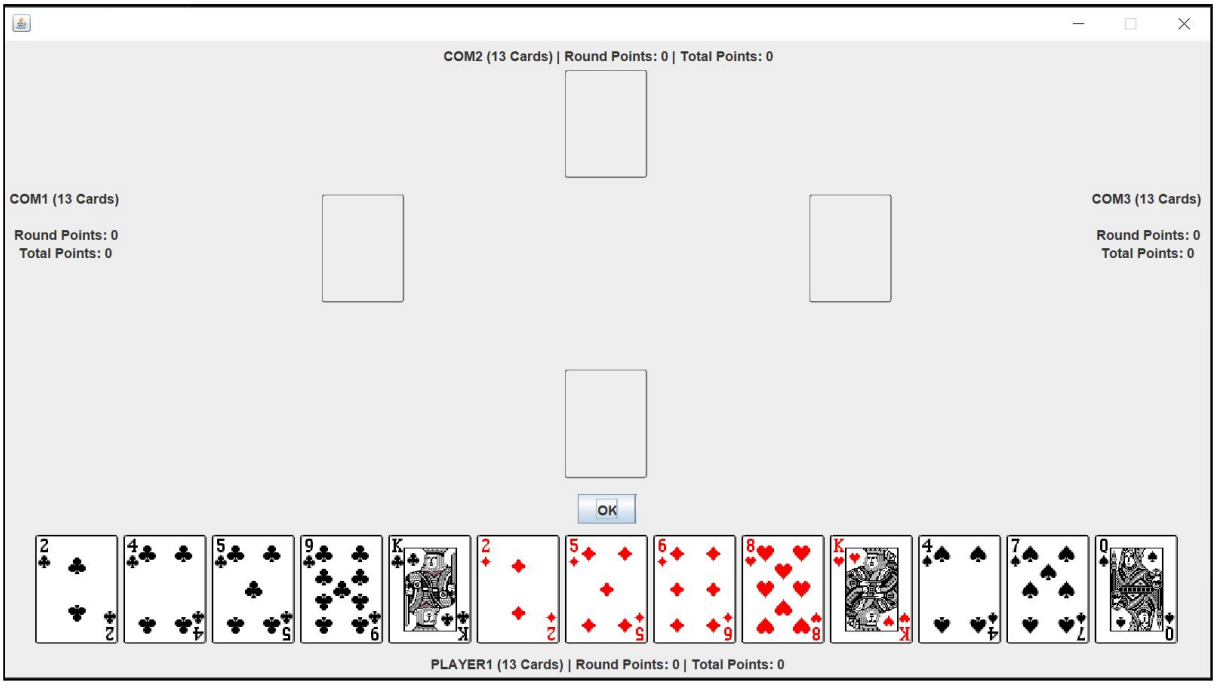
Primary Actor: **Game Controller**
Pre-condition: **Null**
Scenario

- 1 player and 3 computer player will be inserted into the game.
- Main page will be displayed with 52 cards be distributed evenly among the 4 players.
- User will choose 3 cards to pass to the next player on the left and receive 3 cards from computer player on the right.





Player select 3 cards to be pass - 10 clubs, Ace Hearts, 2 Spade.



Player will then receive 3 cards from the ComPlayer - King Clubs, King Hearts, Queen Clubs

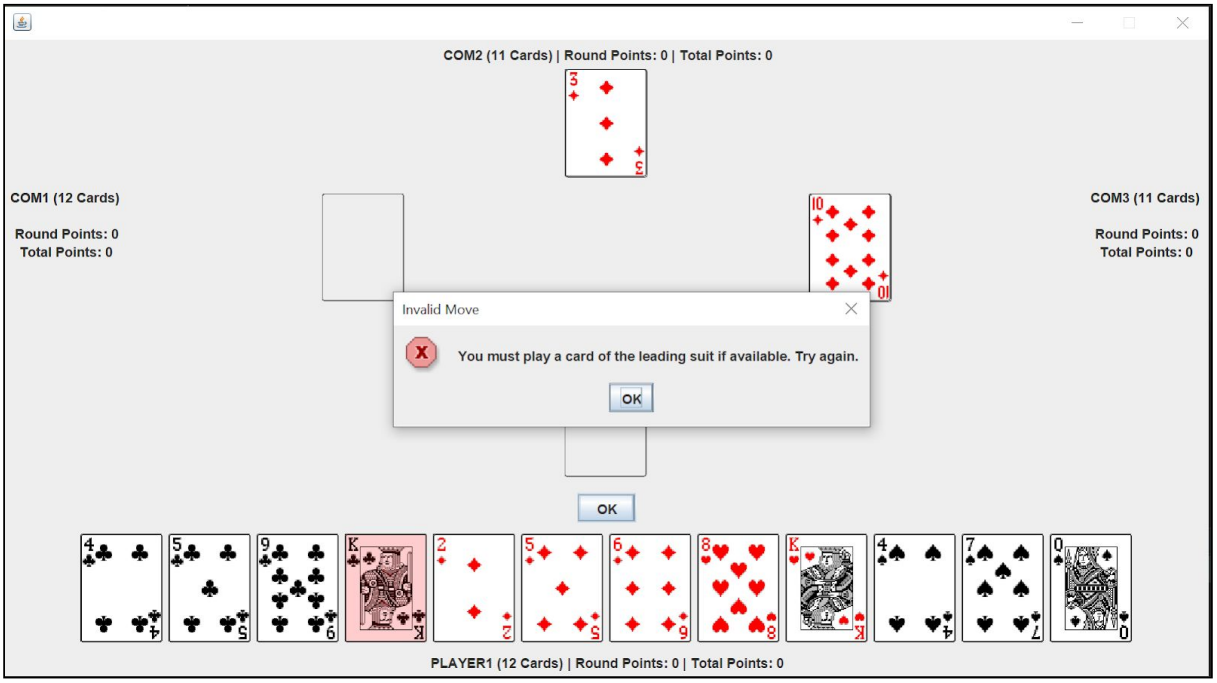
3.1.2. Scenario 2: Set (Each round will consists of 13 Sets)

Primary Actor: **User**

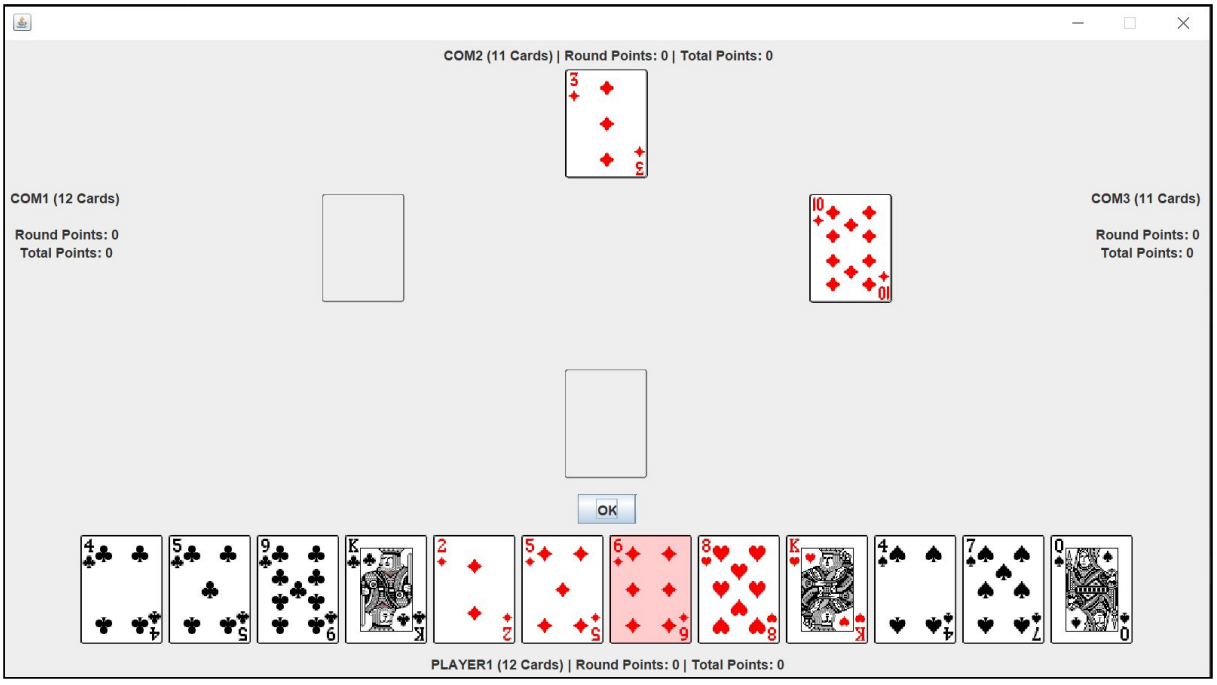
Precondition: **Player to play a card based on the Suit in the Set**

Scenario

- Player will select 2 of Clubs in his hand and if he has one he play the card.
- If player do not have 2 of Clubs, he will play a card depending on the leading suit in the set.
- If player, play a point card in the first set, an error message will be prompted.
- If player, have a leading suit in his hand and he choose to play a non-leading suit card, an error message will be prompted.
- If player, does not have a leading suit he can play any card that is not a point card.
- If player, does not have a leading suit and he only have point card in his hand, he can discards a point card and the hearts is broken.
- If player choose more than 1 cards to play during a set, an error message will be prompted.
- A set will end when all players had played a card.

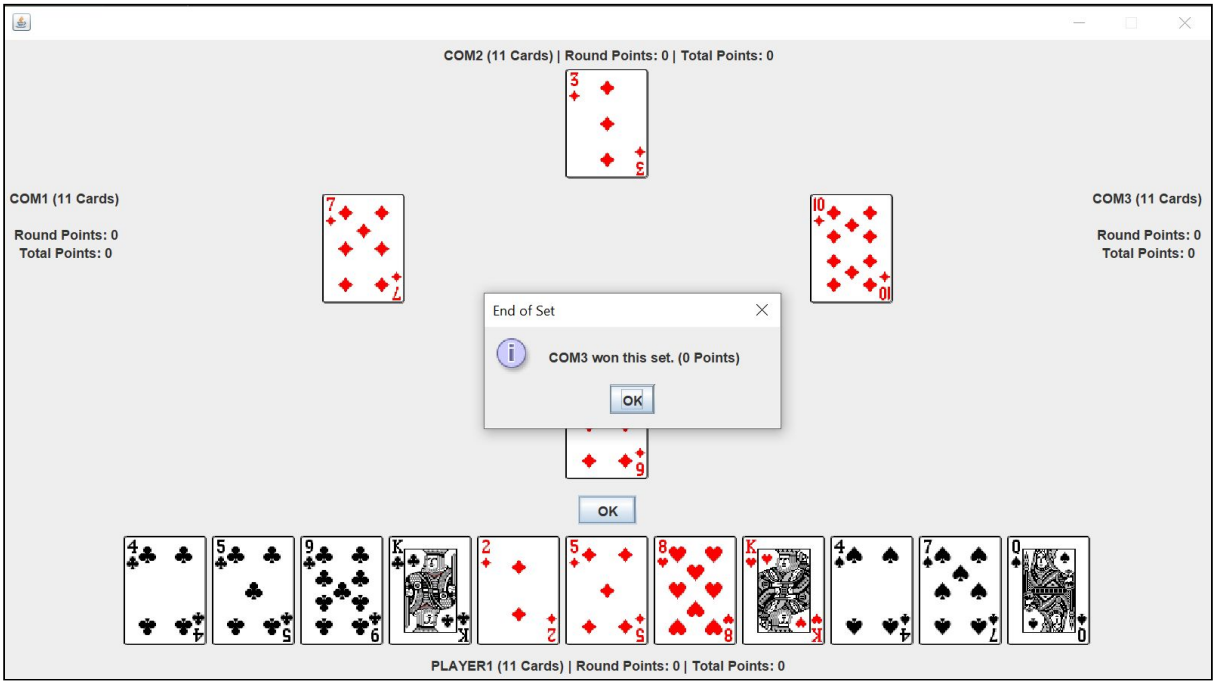


The leading suit is Diamond but player tries to play a King of Clubs an error message will be shown



Player follow the suit and play a 6 of Diamonds

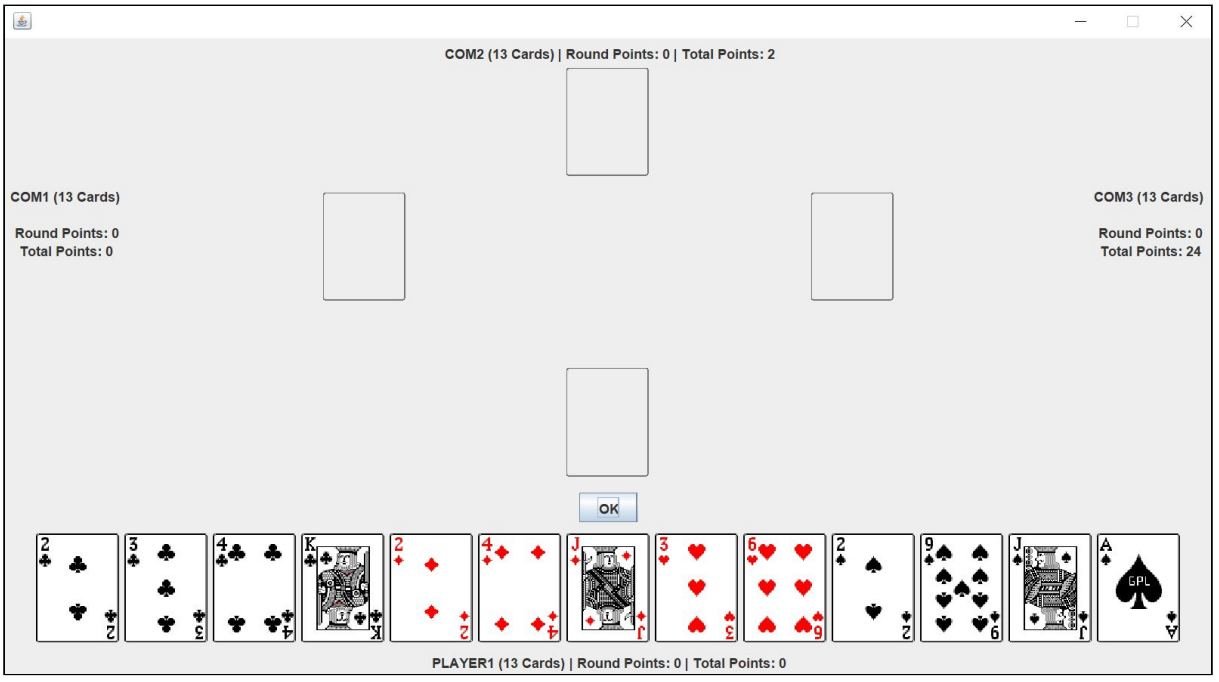
Hi



3.1.3. Scenario 3: Round

Primary Actor: **User**
Precondition: **Game Controller**
Scenario

- Points for players will be accumulated based on individual set scores.
- If the player win the leading suit with a queen of spades - 13 points or any card of suit heart - 1 points.
- If player win 13 hearts and a queen of spades, he has shoot the moon and other player will get 26 points.
- Round will end when all players hand is empty.



The score will be accumulated, in this case COM3 had a total points of 24 after round 1.

3.1.4. Scenario 4: End Game

Primary Actor: **User**
Precondition: **Game Controller**
Scenario

- If a player, reaches 100 points the game will end
- The winner will be the one with the least points.

4. JUNIT Test Case

A unit test is being implemented in the project to test a specific function in the code to check whether it assets a certain behavior or state. We created Junit Test method in a class to test critical methods in our projects. We annotate all test method with the @Test annotation, we uses an assert method to check the expected versus the actual result. We print meaningingful message by displaying the expected and actual output. See below for some example of our test case.

Test 1: tallyPointsForRound Method

This test case will ensure that if a player shoot the moon, the points will be added accordingly.

Expected result: All other player will add 26 points and the player who shoot the moon will not have any points added.

```
@Test
public void tallyPointsForRound(){
    Player[] player = testGame.getListOfPlayers();
    player[0].addToPointsFromCurrentRound(26);
    testGame.tallyPointsForRound();
    assertEquals( expected: 26, player[1].getTotalPoints()); //if shoot the moon the other player will add 26
    assertEquals( expected: 0, player[1].getPointsFromCurrentRound()); //reset round score
}
```

Test 2: getStartPlayerIndex Method

This test case will ensure that the method keep track of the player position that started the set, this method is critical to determine the passing of cards.

Expected result: The index return from this method will be the player that play the card 2 of Clubs.

```
@Test
public void getStartPlayerIndex(){
    GameRegulator regulatorTest = new GameRegulator();
    Game gametest = new Game( numOfPlayers: 4);
    Player[] player = gametest.getListOfPlayers();
    Hand hand1 = player[0].getHand();
    Hand hand2 = player[1].getHand();
    Hand hand3 = player[2].getHand();
    Hand hand4 = player[3].getHand();

    hand1.addCard(new Card(Suit.DIAMONDS,Rank.TWO));
    hand2.addCard(new Card(Suit.HEARTS,Rank.TWO));
    hand3.addCard(new Card(Suit.CLUBS,Rank.TWO));
    hand4.addCard(new Card(Suit.SPADES,Rank.TWO));

    assertEquals( expected: 2, regulatorTest.getStartPlayerIndex(player)); //To test the function to locate the 2 clubs and return the player index
}
```

Test 3: getLeadingSuit Method

This test will check whether the method return the correct suit from the particular set. This will determine the cards that will be played by the player at the later stage of the set.

```
@Test
public void getLeadingSuit() {
    Card card2Spade = new Card(Suit.SPADES,Rank.TWO);
    Card card3Hearts = new Card(Suit.HEARTS, Rank.THREE);
    setTest.addCardToSet(card2Spade,  playerNum: 1);
    setTest.addCardToSet(card3Hearts,  playerNum: 2);
    assertEquals(Suit.SPADES, setTest.getLeadingSuit());
}
```

Test 4: cardRankComparator Method

This test will check the difference between 2 cards and determine which card has a higher precedence over another based on the rank and suit.

```
@Test
public void compare(){
    CardRankComparator c = new CardRankComparator();
    Card card2Hearts = new Card(Suit.HEARTS, Rank.TWO);
    Card card4Hearts = new Card(Suit.HEARTS, Rank.FOUR);
    int result = c.compare(card2Hearts, card4Hearts);
    assertEquals( expected: -2, result); //Test if same suit the difference rank

    Card card5Hearts = new Card(Suit.HEARTS, Rank.FIVE);
    Card card5Spade = new Card(Suit.SPADES, Rank.FIVE);
    int result1 = c.compare(card5Hearts, card5Spade);
    assertEquals( expected: -1, result1); //Test if same rank the different suit
}
```

Test 5: getWinner Method

This test will test the function on how the method determine the winner, based on the logic when a player reaches 100 points and the one with the lowest point will win.

```
@Test
public void getWinner(){
    Player[] player = testGame.getListOfPlayers(); //Test the function on getting the player with the lowest score
    player[0].addToTotalPoints(100);
    player[1].addToTotalPoints(20);
    player[2].addToTotalPoints(10);
    player[3].addToTotalPoints(0);
    assertEquals(player[3].getName(), testGame.getWinner().getName());
}
```

Test 6: choose3CardsToPass Method

This test will test the logic on how the computer will past the 3 cards between player, this particular method encompass many different private methods. Therefore, it is essential to test this method to make sure that there are no fall through errors.

```
@Test
public void choose3CardsToPass() {
    ComPlayer player1 = new ComPlayer( name: "Test");
    Hand hand1 = player1.getHand();
    List<Card> cardList = new ArrayList<>();
    cardList.add(new Card(Suit.SPADES, Rank.THREE));
    cardList.add(new Card(Suit.SPADES, Rank.FOUR));
    cardList.add(new Card(Suit.SPADES, Rank.FIVE));
    cardList.add(new Card(Suit.SPADES, Rank.TWO));
    hand1.addCards(cardList);
    List<Card> cardTestList = player1.choose3CardsToPass();
    assertEquals( expected: 3, cardTestList.size()); //Test whether it remove 3 cards
    assertEquals(cardTestList.get(2), hand1.getCardsSortedByRank().get(cardList.size() - 1));
}
```

Test 7: dealCard Method

This test will test the dealing of card and ensure that the desk is empty after dealing and each players are given 13 cards.

```
@Test
public void dealCard() {
    assertEquals( expected: 52,decktest.getSizeOfDeck());
    assertEquals( expected: 52, decktest.getSizeOfDeck());

    while (!decktest.isEmpty()) {
        decktest.dealCard();
    }
        assertNull(decktest.dealCard());
    }
```

Test 8: chooseCardToPlay Method

This is to test the logic of the computer player for each set, the computer will play the correct card accordingly based on the card on his hand.

```
@Test
public void chooseCardToPlay() {
    ComPlayer player1 = new ComPlayer( name: "Test");
    Hand hand1 = player1.getHand();
    List<Card> cardList = new ArrayList<>();
    cardList.add(new Card(Suit.SPADES, Rank.THREE));
    cardList.add(new Card(Suit.HEARTS, Rank.FIVE));
    hand1.addCards(cardList);

    Set round1 = new Set();
    round1.addCardToSet(new Card(Suit.SPADES, Rank.TWO), playerNum: 1);
    round1.addCardToSet(new Card(Suit.SPADES, Rank.FOUR), playerNum: 1);

    Set round2 = new Set();

    Set round3 = new Set();
    round1.addCardToSet(new Card(Suit.SPADES, Rank.TWO), playerNum: 1);
    round1.addCardToSet(new Card(Suit.SPADES, Rank.FOUR), playerNum: 1);
    round1.addCardToSet(new Card(Suit.SPADES, Rank.THREE), playerNum: 1);

    assertEquals(player1.chooseCardToPlay(round3, isHeartsBroken: true), cardList.get(0));
    assertEquals(player1.chooseCardToPlay(round2, isHeartsBroken: false), cardList.get(0));
    assertEquals(player1.chooseCardToPlay(round1, isHeartsBroken: false), cardList.get(0));
}
```

Test 9: addCard Method

This is to test whether the when added new card will the size of the deck increase and if the deck already have 52 cards can the method still allow adding of cards.

```
@Test
public void addCard(){
    List<Card> list = new ArrayList<>();
    list = decktest.getDeck();
    assertEquals( expected: 52, decktest.getSizeOfDeck());
    Card c1 = new Card(Suit.HEARTS, Rank.ACE);
    Card card = list.get(0);
    list.remove(card);
    assertEquals( expected: 51, list.size());
    decktest.addCard(c1);
    assertEquals( expected: 52, decktest.getSizeOfDeck());
}
```

Test 10: removeCards Method

Test if the function can successfully remove a list of cards from its current deck and after removing does the number of cards tally.

```
@Test
public void removeCards() {
    assertEquals( expected: 0, handTest.getNumberOfCards());
    List<Card> list = new Deck().getDeck();
    handTest.addCards(list);
    assertEquals( expected: 52, handTest.getNumberOfCards());

    List<Card> cardsToRemove = new ArrayList<>();
    cardsToRemove.add(new Card(Suit.HEARTS, Rank.TWO));
    cardsToRemove.add(new Card(Suit.CLUBS, Rank.TWO));
    cardsToRemove.add(new Card(Suit.SPADES, Rank.TWO));
    handTest.removeCards(cardsToRemove); //remove 3 cards

    assertEquals( expected: 49, handTest.getNumberOfCards());
}
```

5. Conclusion

This project was an enriching and fulfilling experience which provided us with a platform for us to delve deeper into the concepts of object-oriented programming (OOP). Through it, we have realised the effectiveness and efficiency of creating applications in an object-oriented structure. Not only that, it also provided us with an opportunity to apply what we have learnt in class in a more practical way and to learn more about implementing Java GUIs. Though we were unfortunately unable to implement a client-server application that supports up to 4 human players, we look forward to continuing our project to make that a reality.

Reference List

Design Patterns Builder Pattern. (n.d). Retrieved from https://www.tutorialspoint.com/design_pattern/builder_pattern.htm

Unit Testing with JUnit - Tutorial. (2007). Retrieved from https://www.vogella.com/tutorials/JUnit/article.html