SYST 17796 Deliverable 2

DESIGN DOCUMENT TEMPLATE

Overview

1. Project Background and Description

As in deliverable 1, we have mentioned that we are selecting a Card War Game for this project.

The rules of our Card War Game were already described in the last deliverable. In this game, the joker will be removed from the deck, leaving us with just 52 cards to use. Additionally, this game can only currently be played between two people. Although this can be played with two or more players, we are currently creating it with two players in mind. The functionality of this game can be expanded in the future to accommodate more players.

Since there are 52 cards in the deck, each of the two players will start the game with 26–26 cards each. The person who has the most cards in hand at the end of the game is declared the winner. The primary game rule is already covered in Deliverable 1 of this project. Now that the use case for the game has been added, along with a use case diagram, please refer to Figure No. 1.

Use Case:

1. Shuffle a 52-card deck.
2. Deal two players a face-down deck of cards.
3. The highest card from each player's deck will be displayed.
4. The person holding the higher card places both cards at the bottom of their pile.
5. Go back to step 3.
   Alternative : War
   In step 4, If both the cards have the same value.
6. Declare: "war".
7. In this situation, three cards will be dealt face down to each participant.
8. Go back to step 3.
   Alternative : if there are insufficient cards for war.
   At step 7, a player does not have three cards.
9. Draw all cards in your hand except one (for step 3).
10. If this player is out of cards, , put the top card back (for step 3).
11. Go back to step 3.
    Alternative:
    At step 5, the player runs out of cards .
    The game is won by another player.

Use Case Diagram:

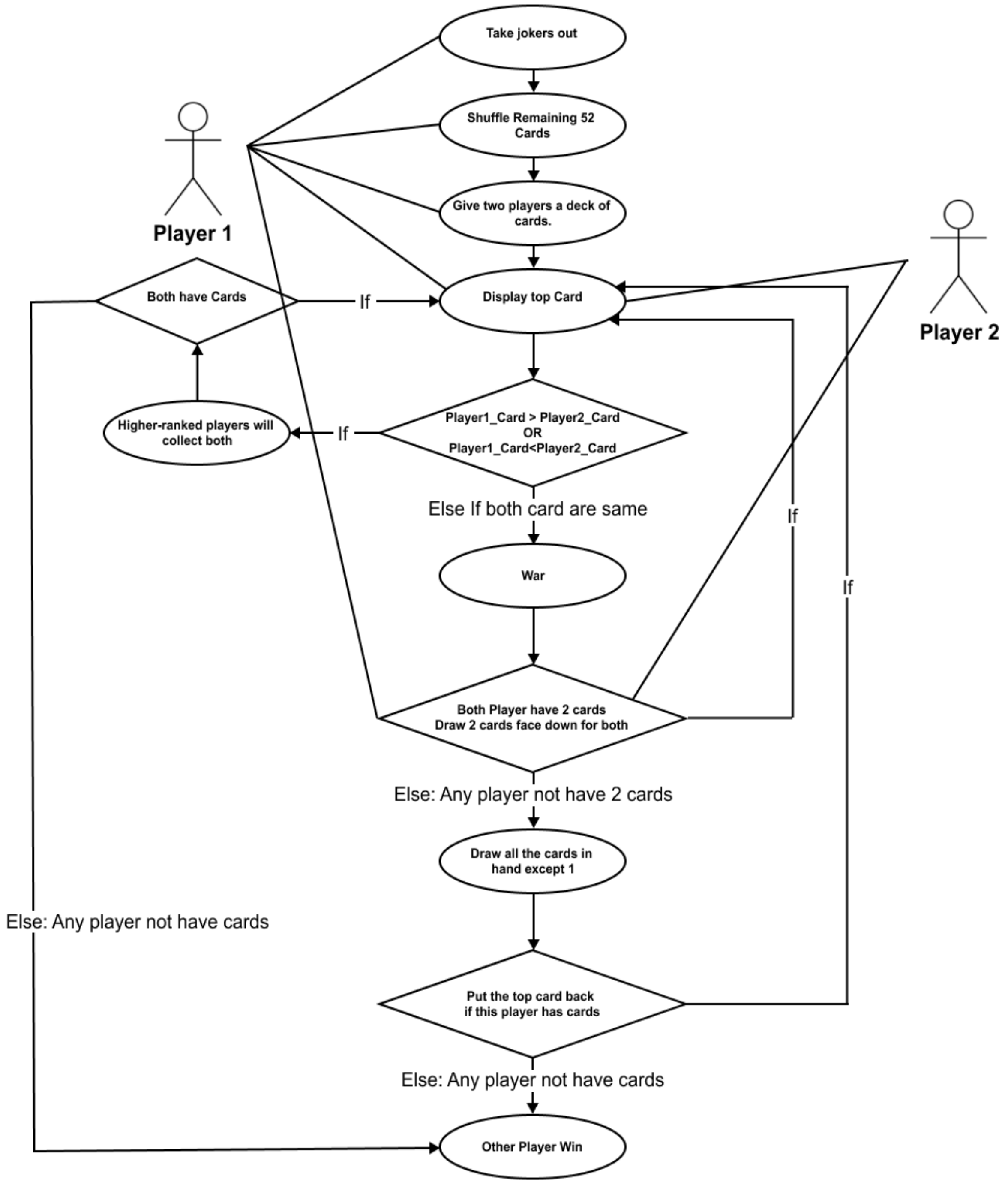
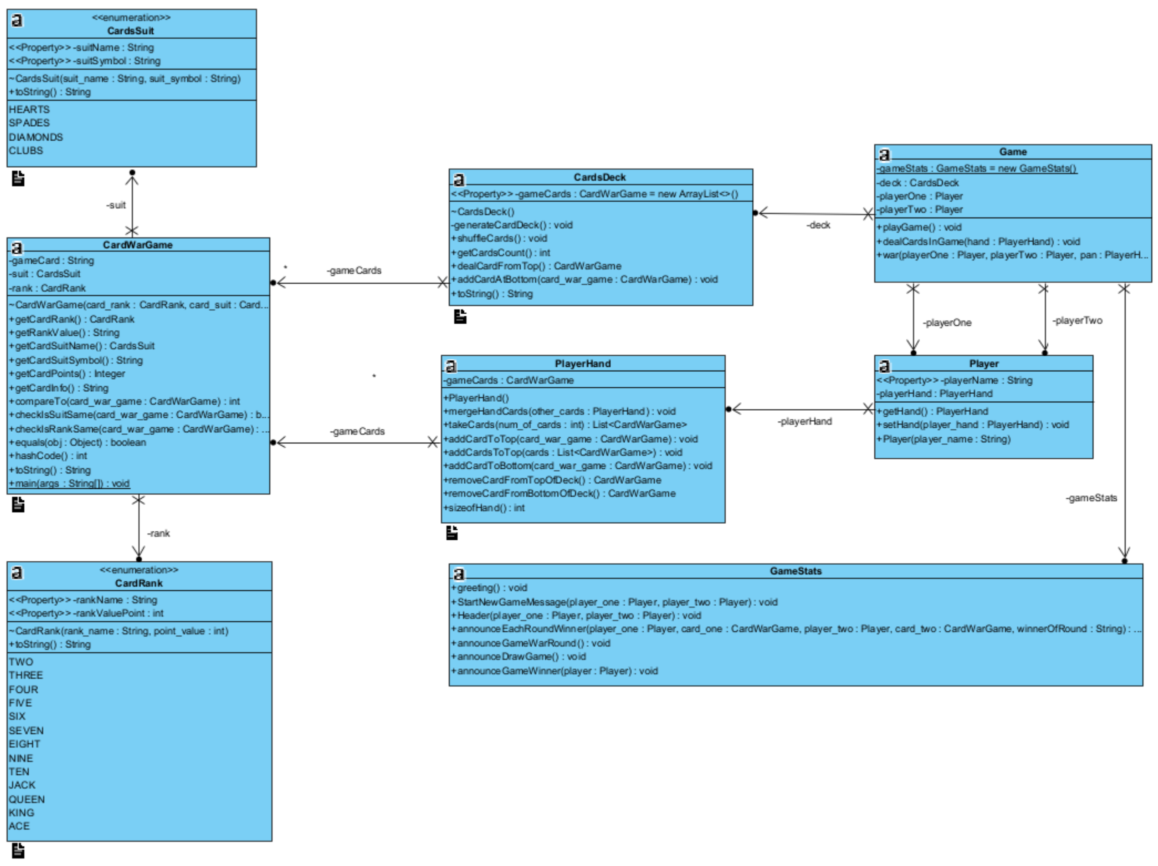

Take jokers out
Shuffle Remaining 52 Cards
Give two players a deck of cards.
Player 1
Player 2
Both have Cards
If
Display top Card
Higher-ranked players will collect both
If
Player1_Card > Player2_Card OR Player1_Card<Player2_Card
Else If both card are same
War
If
If
Both Player have 2 cards Draw 2 cards face down for both
Else: Any player not have 2 cards
Draw all the cards in hand except 1
Else: Any player not have cards
Put the top card back if this player has cards
Else: Any player not have cards
Other Player Win

Figure 1

2. Design Considerations

Class Diagram:



<<enumeration>>
CardsSuit
<<Property>> -suitName : String
<<Property>> -suitSymbol : String
~CardsSuit(suit_name : String, suit_symbol : String)
+toString() : String
HEARTS
SPADES
DIAMONDS
CLUBS
-suit
CardWarGame
-gameCard : String
-suit : CardsSuit
-rank : CardRank
~CardWarGame(card_rank : CardRank, card_suit : Card...
+getCardRank() : CardRank
+getRankValue() : String
+getCardSuitName() : CardsSuit
+getCardSuitSymbol() : String
+getCardPoints() : Integer
+getCardInfo() : String
+compareTo(card_war_game : CardWarGame) : int
+checkIsSuitSame(card_war_game : CardWarGame) : b...
+checkIsRankSame(card_war_game : CardWarGame) : ...
+equals(obj : Object) : boolean
+hashCode() : int
+toString() : String
+main(args : String[]) : void
-rank
<<enumeration>>
CardRank
<<Property>> -rankName : String
<<Property>> -rankValuePoint : int
~CardRank(rank_name : String, point_value : int)
+toString() : String
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN
JACK
QUEEN
KING
ACE
-gameCards
CardsDeck
<<Property>> -gameCards : CardWarGame = new ArrayList<>()
~CardsDeck()
-generateCardDeck() : void
+shuffleCards() : void
+getCardsCount() : int
+dealCardFromTop() : CardWarGame
+addCardAtBottom(card_war_game : CardWarGame) : void
+toString() : String
-deck
Game
-gameStats : GameStats = new GameStats()
-deck : CardsDeck
-playerOne : Player
-playerTwo : Player
+playGame() : void
+dealCardsInGame(hand : PlayerHand) : void
+war(playerOne : Player, playerTwo : Player, pan : PlayerH...
-playerOne
-playerTwo
-gameCards
PlayerHand
-gameCards : CardWarGame
+PlayerHand()
+mergeHandCards(other_cards : PlayerHand) : void
+takeCards(num_of_cards : int) : List<CardWarGame>
+addCardToTop(card_war_game : CardWarGame) : void
+addCardsToTop(cards : List<CardWarGame>) : void
+addCardToBottom(card_war_game : CardWarGame) : void
+removeCardFromTopOfDeck() : CardWarGame
+removeCardFromBottomOfDeck() : CardWarGame
+sizeofHand() : int
-playerHand
Player
<<Property>> -playerName : String
-playerHand : PlayerHand
+getHand() : PlayerHand
+setHand(player_hand : PlayerHand) : void
+Player(player_name : String)
-gameStats
GameStats
+greeting() : void
+StartNewGameMessage(player_one : Player, player_two : Player) : void
+Header(player_one : Player, player_two : Player) : void
+announceEachRoundWinner(player_one : Player, card_one : CardWarGame, player_two : Player, card_two : CardWarGame, winnerOfRound : String) : ...
+announceGameWarRound() : void
+announceDrawGame() : void
+announceGameWinner(player : Player) : void

Figure 2

On the basis of the given use case diagram, we have created 8 classes, as shown in the above class diagram (Figure 2): CardWarGame, CardsDeck, Game, CardsSuit, CardRank, PlayerHand, Player, and GameStats. There will be a set of methods and parameters for each class.

Encapsulation

In our project, we use encapsulation to hide the implementation details of each card and only expose public methods that allow the game logic to interact with them, such as methods for getting the card's value or suit.

Delegation

In our project, delegation can be used in the Game class, which gives the CardsDeck class the responsibility of dealing cards.

Cohesion

We use cohesion to design each component to handle a specific aspect of the game logic, such as a component for managing the deck of cards, another for managing the player's hands, and another for managing the rules of the game.

Coupling

Coupling can be minimised by designing each component to depend simply on the interfaces of other components, rather than on their internal implementation details.

Inheritance

We will not be directly using **inheritance** in this project but we will use the function of once to other classes by creating objects.

Aggregation

We use aggregation to combine multiple player objects into a game object that manages the overall state of the game.

Composition

We will use composition to create a deck of cards by combining individual card objects.

Flexibility/Maintainability

If we will talk about the **flexibility** of this project then we are keeping a scope so that we can extend it in the future for more players than 2. In terms of **Maintainability**, this project is not going to require much maintainability as we have already handled all possible cases in the case diagram with the help of an alternate path