IS442: Object Oriented Programming

Project: Estimation Card Game

Analysis and Design

G1T12

Group Members:

Lim Khong Mun Elias

Madhumitha D/O Suresh Kumar

Wong Xian Rui Abel

Contents

Design Pattern and Principles

Design patterns considered and utilised

Model View Controller

For the overall game design pattern, we considered using the MVC pattern as it was the one, we were most familiar with. We started building our application with the intention that it would be a console application. Thus, gameLogic class would act as the controller, with the other models being the player and scoreboard classes.

Once we built up our console application, we started breaking down the 2 console methods (startRound and startSubRound) within the controller and rebuilding it as a view.

It was difficult at first as we didn't have experience and assumed that converting the System.out.println() outputs as well as the scanner inputs to GUI would be straightforward and seamless. We wanted the game to be run through the GameLogic class which calls the GUI methods. We decided to run the game logic through the GameUI class itself and use GameLogic class as a controller to manipulate the model which updates the GUI. GameUI calls the necessary methods in GameLogic class which is activated by the different buttons which are attached to action listeners. The actions include starting the game, selecting how many tricks to win, and cards to play. GameLogic class accepts the action events as input and converts it to commands for the game.

Within the GameLogic class we separated the entire logic of the game into different methods which are executed within a while loop. This loop, todoThread(), controls the sequence of events that should be executed when the game is played. We simply check for 3 different things: whether a subround has been completed, whether a round has been completed, and whether the game is completed. The main method where the game play is executed happens in playSubRound method.

Programming to an Interface, not an Implementation

Throughout our models, we also applied the concept of programming to an interface. We were provided with the Hand interface, which we used to create our PlayerHand within the game. Although it did allow us to enforce methods which we needed (and were identical to families of objects which were Hand-like (like tableHand)), it also resulted in excess methods. This meant that methods we did not utilise like evaluateHand needed to be overridden too.

This could simply be a result of the scope of the game being too small such that it did not require the particular method. However, the other thought we had was that perhaps Hand contained far too many abstract methods that were unnecessary as an interface. This conflicted with the Interface Segregation Principle which we tried to abide by too.

Composition
Since we needed dynamically changing objects at run time through objects acquiring references to other objects, the majority of the logic in our console logic and subsequently our views were designed with object composition in mind.

However, we realised a key disadvantage to this is that because an object's implementation is written with object interfaces, there were times when we passed in incorrect objects which resulted in errors, simply because the objects had the same interfaces.

This was swiftly addressed by ensuring that each class in our models were encapsulated and focused solely on a single use case. This then prevented any major spaghetti code once the logic of the game started fleshing out.

In all, composition was an excellent design pattern and the key to allowing us to assemble together existing components and creating our GUI application from a console app. Without this design pattern, it would have been a hassle to reuse our classes.

Delegation
We utilised delegation when crafting objects such as our TableHand, which is essentially just an ArrayList of Objects. Since we needed to reuse the behaviour of an ArrayList (which is very much similar like a Table, as we could get cards from a table, clear the table, etc.) while having a class contain 2 instances of an ArrayList, delegation made sense.

Because our parameters in the TableHand were rather straight forward (card and player), it did not complicate and make things more difficult to understand. Thus, it was a good principle that allowed us to carry out our functionality more easily as opposed to implementing hand together with tableHand and making life more difficult for ourselves.

Design Principles Considered

Besides the core OOP principles of abstraction, encapsulation, polymorphism and inheritance, we read up and utilised a bunch of other principles.

Don't Repeat Yourself (DRY)
Thankfully with the help of IntelliJ, repeat code was minimised for our team. We wrote helper methods to reduce the repetition of code and included hardcoded values for the rounds and surrounds (due to its many repetitions).

Encapsulate What Changes
This was especially important because our logic for the computer's algorithm and bid was constantly changing due to our test cases failing. As such it was good that we encapsulated the computer's method which prevented our game from crashing when certain variables changed.

Single Responsibility Principle (SRP)
We tried as much as possible to leave only one functionality for one class in order to reduce coupling between two or more functionalities. This served us well as we relied a lot on composition in our main GUI view methods and prevented the game from crashing unnecessarily.

Favour Composition Over Inheritance
Most of our methods were built using composition instead of inheritance due to the flexibility that it brought. This brings us to our next point on the interface segregation principle.

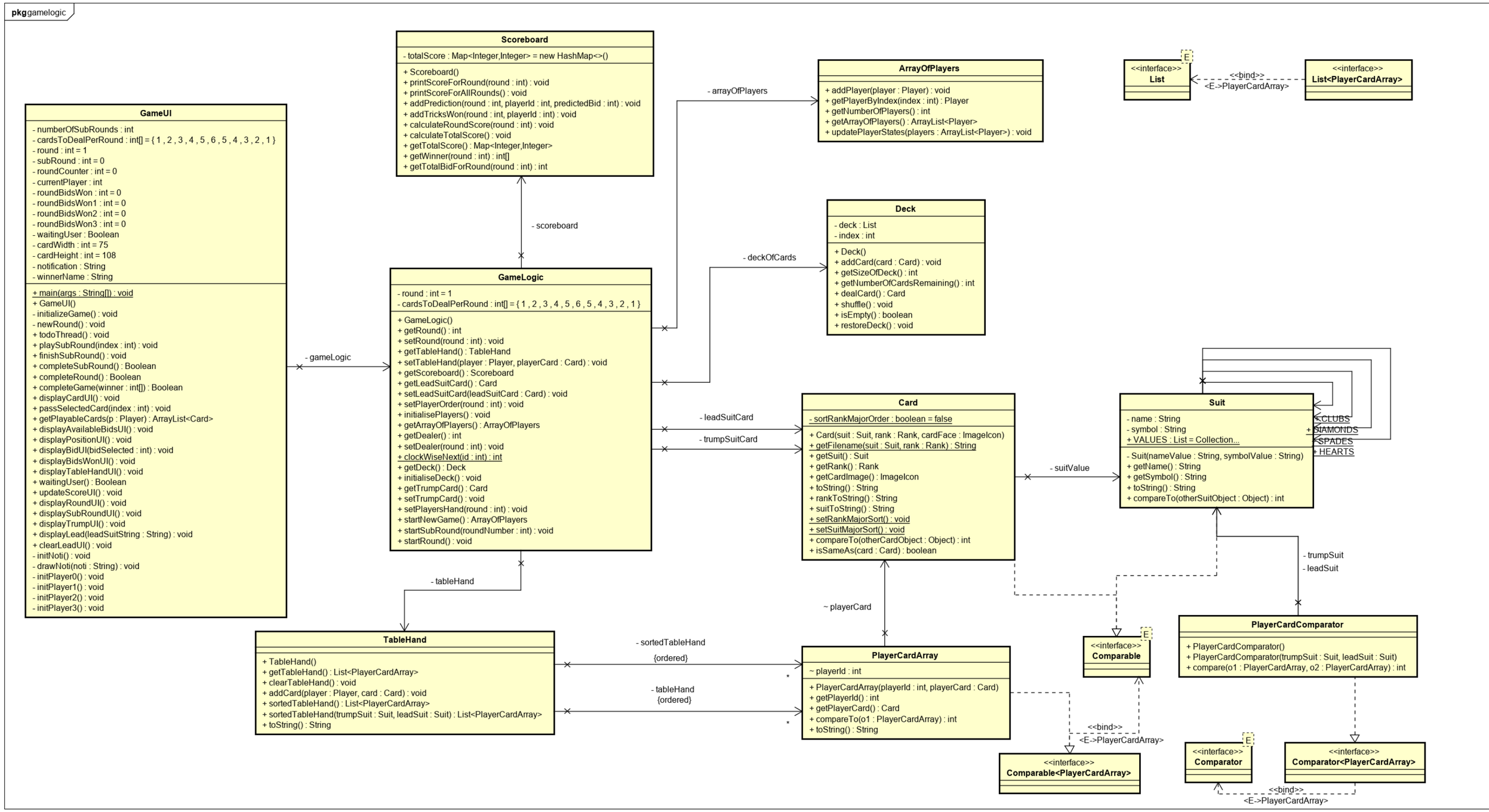
Interface Segregation Principle
Here, we realised that we should not have implemented an interface if we did not use all of the methods that it offered. Back when we were coding, we assumed that overriding the method with a dummy variable would be fine. But perhaps this could be due to the small scope of our application. The dummy variable would clearly break a future feature once the application grows larger in future.

Coding to Interface and not Implementation
Programming to an interface and not an implementation was important to hide the things we did not need to know and provides a schema on how the object will behave.

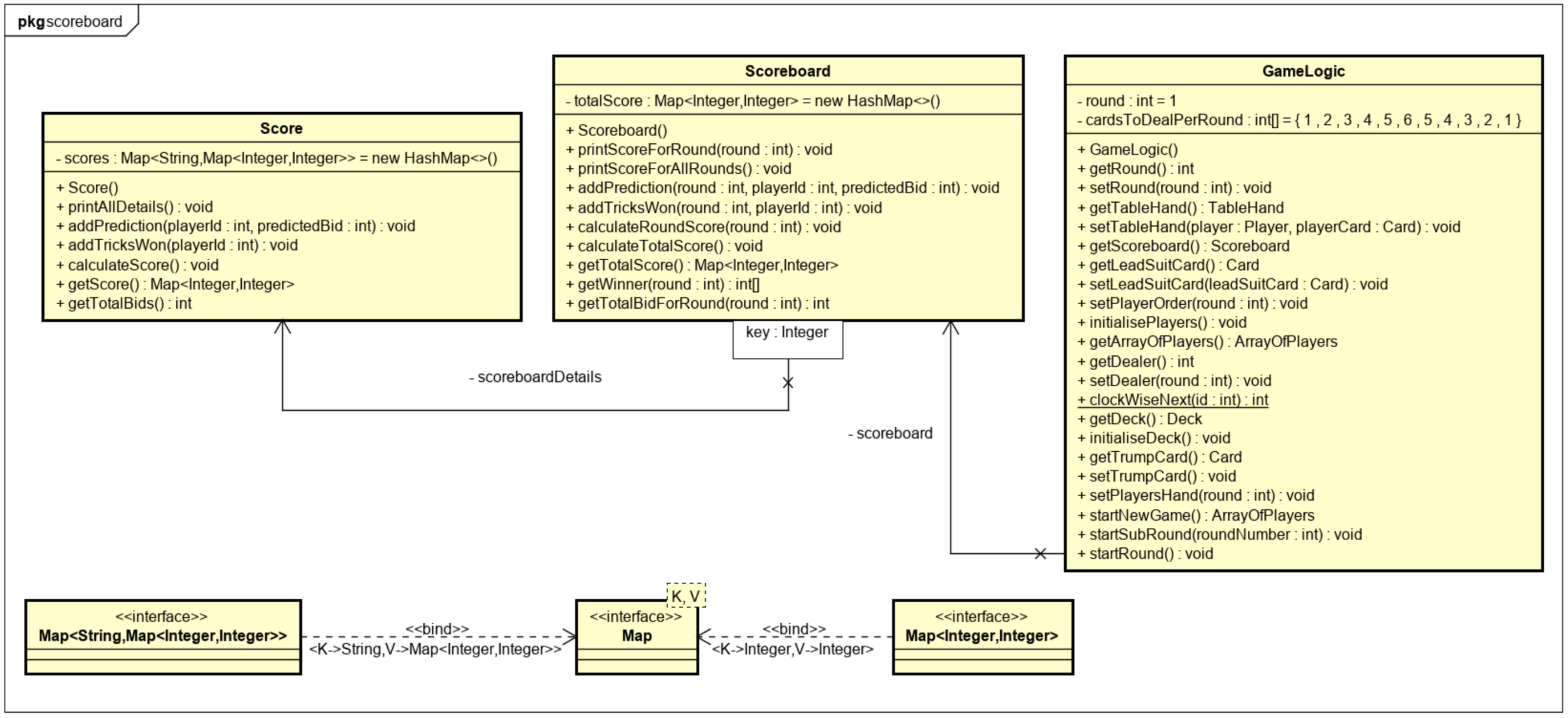
Diagrams

UML Diagrams

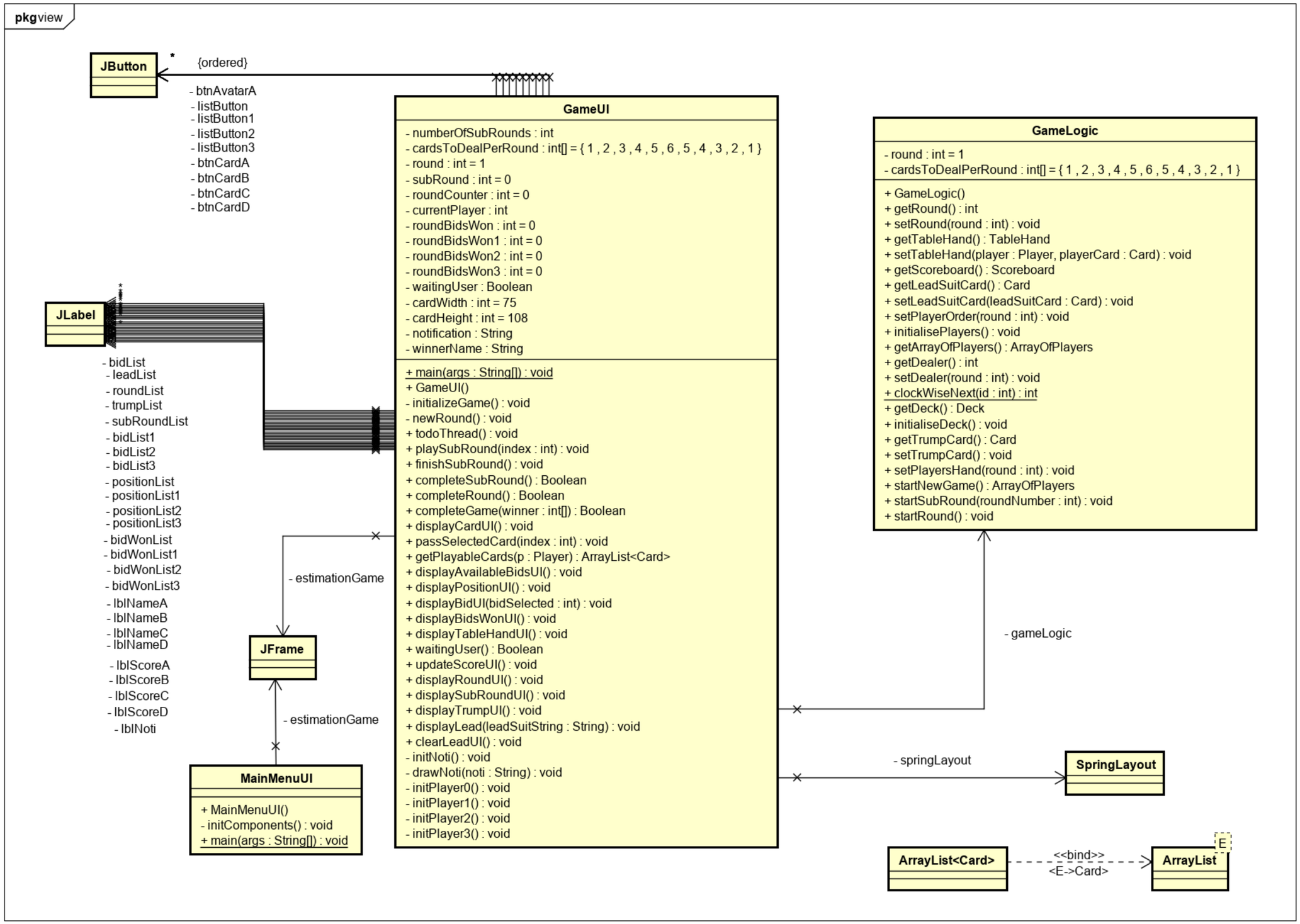
Game Logic



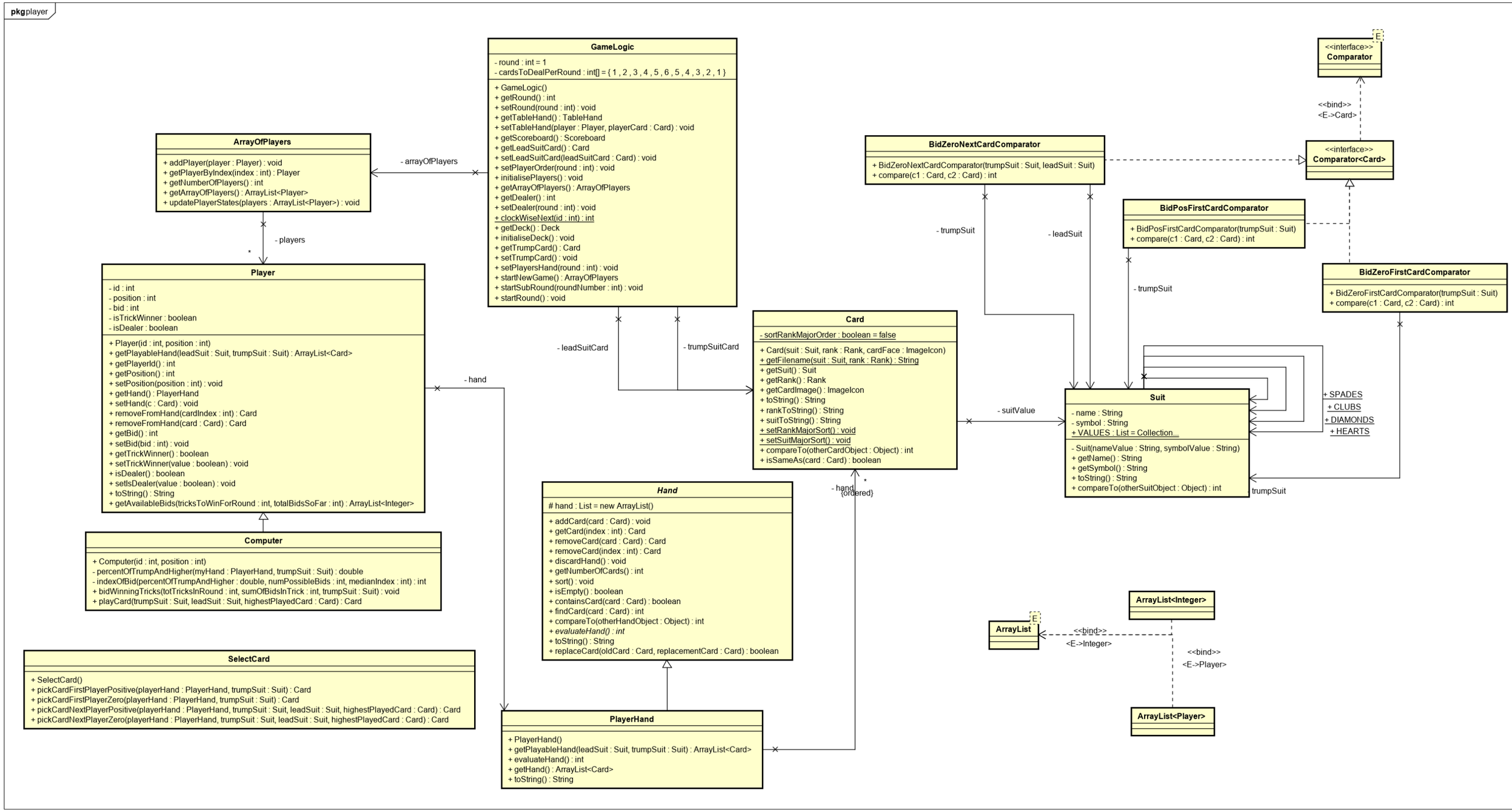
Scoreboard


pkg scoreboard
Score
- scores : Map<String,Map<Integer,Integer>> = new HashMap<>()
+ Score()
+ printAllDetails() : void
+ addPrediction(playerId : int, predictedBid : int) : void
+ addTricksWon(playerId : int) : void
+ calculateScore() : void
+ getScore() : Map<Integer,Integer>
+ getTotalBids() : int
Scoreboard
- totalScore : Map<Integer,Integer> = new HashMap<>()
+ Scoreboard()
+ printScoreForRound(round : int) : void
+ printScoreForAllRounds() : void
+ addPrediction(round : int, playerId : int, predictedBid : int) : void
+ addTricksWon(round : int, playerId : int) : void
+ calculateRoundScore(round : int) : void
+ calculateTotalScore() : void
+ getTotalScore() : Map<Integer,Integer>
+ getWinner(round : int) : int[]
+ getTotalBidForRound(round : int) : int
key : Integer
- scoreboardDetails
- scoreboard
GameLogic
- round : int = 1
- cardsToDealPerRound : int[] = { 1 , 2 , 3 , 4 , 5 , 6 , 5 , 4 , 3 , 2 , 1 }
+ GameLogic()
+ getRound() : int
+ setRound(round : int) : void
+ getTableHand() : TableHand
+ setTableHand(player : Player, playerCard : Card) : void
+ getScoreboard() : Scoreboard
+ getLeadSuitCard() : Card
+ setLeadSuitCard(leadSuitCard : Card) : void
+ setPlayerOrder(round : int) : void
+ initialisePlayers() : void
+ getArrayOfPlayers() : ArrayOfPlayers
+ getDealer() : int
+ setDealer(round : int) : void
+ clockWiseNext(id : int) : int
+ getDeck() : Deck
+ initialiseDeck() : void
+ getTrumpCard() : Card
+ setTrumpCard() : void
+ setPlayersHand(round : int) : void
+ startNewGame() : ArrayOfPlayers
+ startSubRound(roundNumber : int) : void
+ startRound() : void
<<interface>>
Map<String,Map<Integer,Integer>>
<<bind>>
<K->String,V->Map<Integer,Integer>>
K, V
<<interface>>
Map
<<bind>>
<K->Integer,V->Integer>
<<interface>>
Map<Integer,Integer>

Views

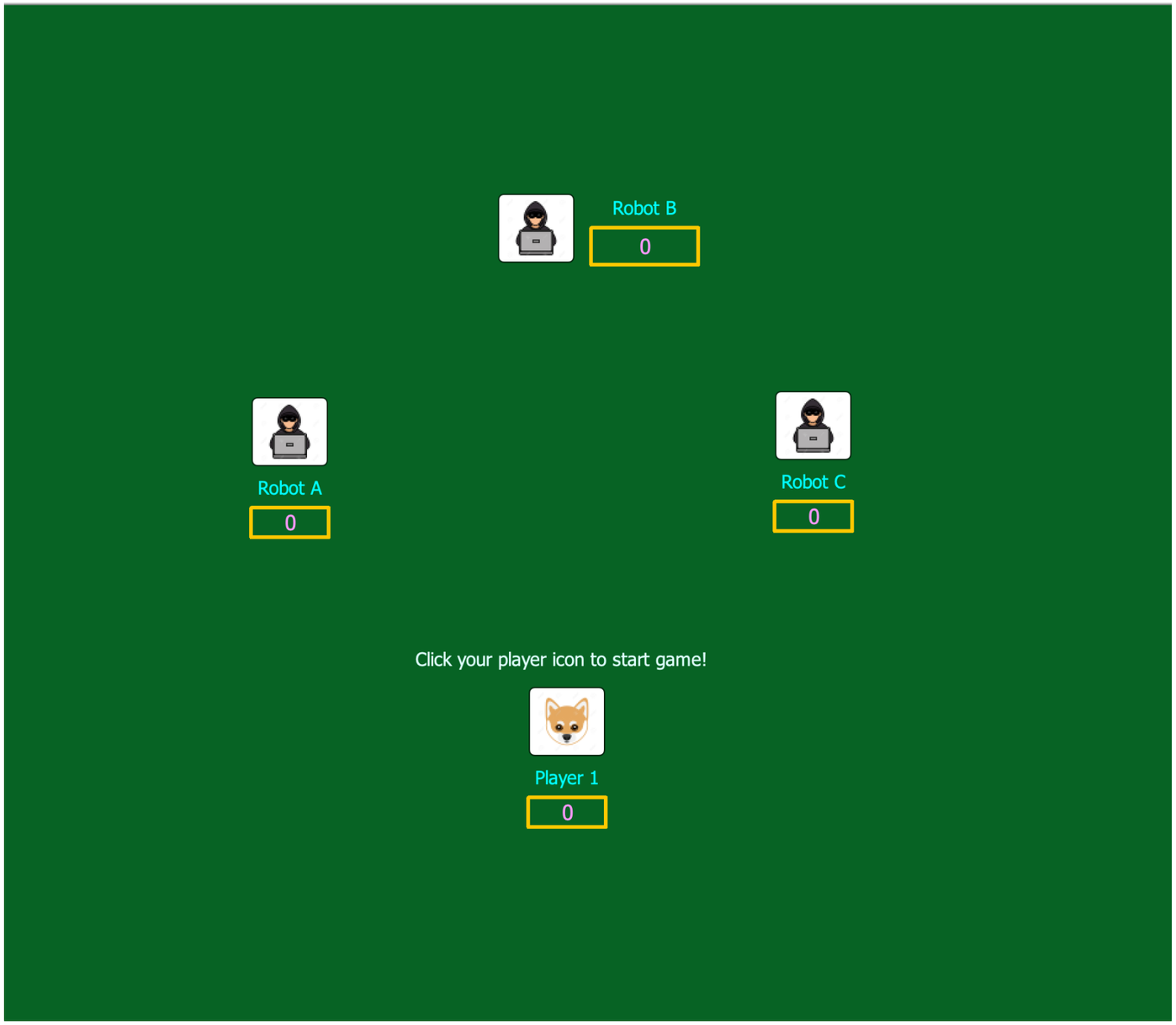

pkg view
JButton
{ordered}
- btnAvatarA
- listButton
- listButton1
- listButton2
- listButton3
- btnCardA
- btnCardB
- btnCardC
- btnCardD
JLabel
- bidList
- leadList
- roundList
- trumpList
- subRoundList
- bidList1
- bidList2
- bidList3
- positionList
- positionList1
- positionList2
- positionList3
- bidWonList
- bidWonList1
- bidWonList2
- bidWonList3
- lblNameA
- lblNameB
- lblNameC
- lblNameD
- lblScoreA
- lblScoreB
- lblScoreC
- lblScoreD
- lblNoti
GameUI
- numberOfSubRounds : int
- cardsToDealPerRound : int[] = { 1 , 2 , 3 , 4 , 5 , 6 , 5 , 4 , 3 , 2 , 1 }
- round : int = 1
- subRound : int = 0
- roundCounter : int = 0
- currentPlayer : int
- roundBidsWon : int = 0
- roundBidsWon1 : int = 0
- roundBidsWon2 : int = 0
- roundBidsWon3 : int = 0
- waitingUser : Boolean
- cardWidth : int = 75
- cardHeight : int = 108
- notification : String
- winnerName : String
+ main(args : String[]) : void
+ GameUI()
- initializeGame() : void
- newRound() : void
+ todoThread() : void
+ playSubRound(index : int) : void
+ finishSubRound() : void
+ completeSubRound() : Boolean
+ completeRound() : Boolean
+ completeGame(winner : int[]) : Boolean
+ displayCardUI() : void
+ passSelectedCard(index : int) : void
+ getPlayableCards(p : Player) : ArrayList<Card>
+ displayAvailableBidsUI() : void
+ displayPositionUI() : void
+ displayBidUI(bidSelected : int) : void
+ displayBidsWonUI() : void
+ displayTableHandUI() : void
+ waitingUser() : Boolean
+ updateScoreUI() : void
+ displayRoundUI() : void
+ displaySubRoundUI() : void
+ displayTrumpUI() : void
+ displayLead(leadSuitString : String) : void
+ clearLeadUI() : void
- initNoti() : void
- drawNoti(noti : String) : void
- initPlayer0() : void
- initPlayer1() : void
- initPlayer2() : void
- initPlayer3() : void
GameLogic
- round : int = 1
- cardsToDealPerRound : int[] = { 1 , 2 , 3 , 4 , 5 , 6 , 5 , 4 , 3 , 2 , 1 }
+ GameLogic()
+ getRound() : int
+ setRound(round : int) : void
+ getTableHand() : TableHand
+ setTableHand(player : Player, playerCard : Card) : void
+ getScoreboard() : Scoreboard
+ getLeadSuitCard() : Card
+ setLeadSuitCard(leadSuitCard : Card) : void
+ setPlayerOrder(round : int) : void
+ initialisePlayers() : void
+ getArrayOfPlayers() : ArrayOfPlayers
+ getDealer() : int
+ setDealer(round : int) : void
+ clockWiseNext(id : int) : int
+ getDeck() : Deck
+ initialiseDeck() : void
+ getTrumpCard() : Card
+ setTrumpCard() : void
+ setPlayersHand(round : int) : void
+ startNewGame() : ArrayOfPlayers
+ startSubRound(roundNumber : int) : void
+ startRound() : void
- estimationGame
JFrame
- estimationGame
MainMenuUI
+ MainMenuUI()
- initComponents() : void
+ main(args : String[]) : void
- gameLogic
- springLayout
SpringLayout
ArrayList<Card>
<<bind>>
<E->Card>
ArrayList
E

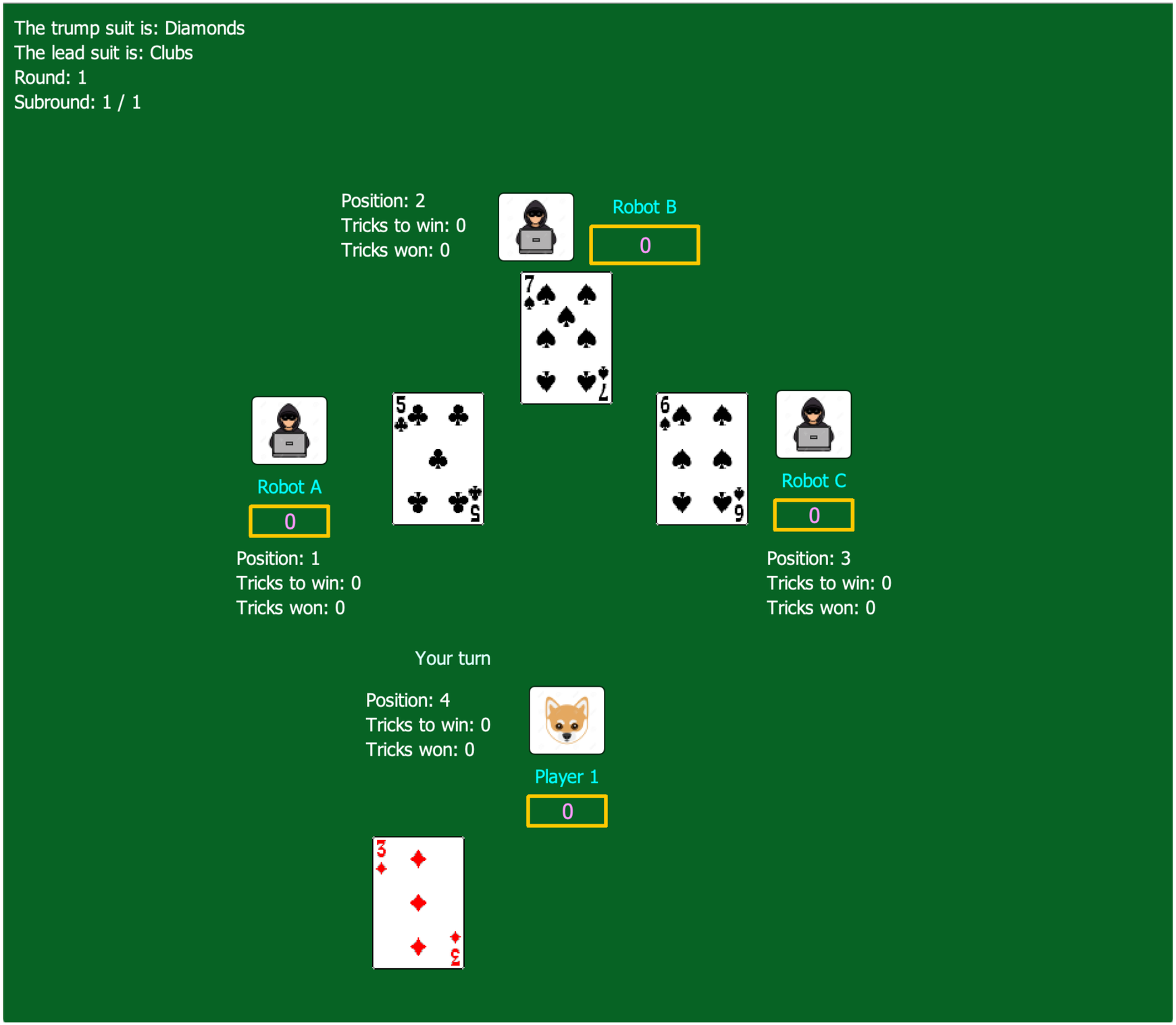
Player

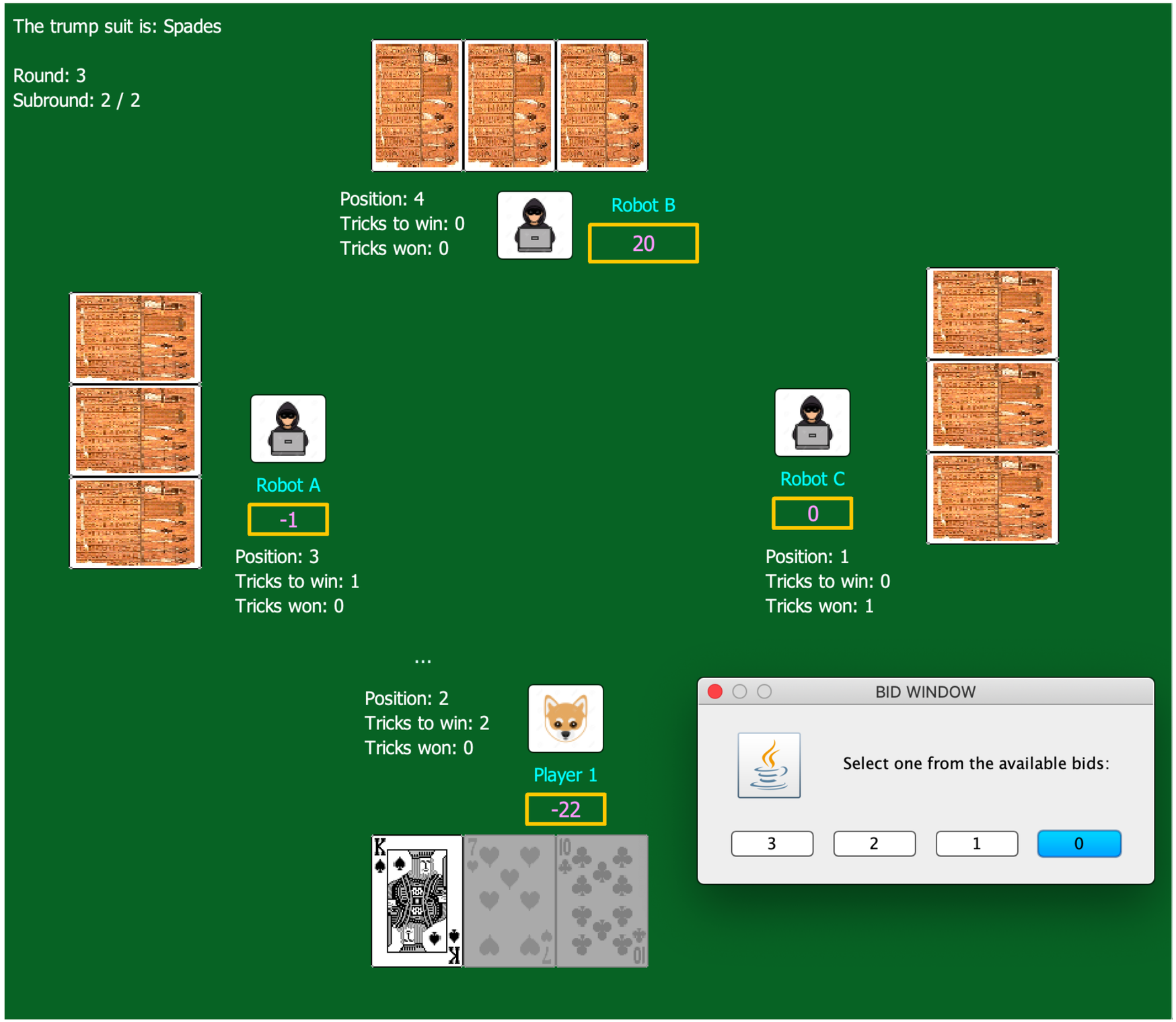

pkg player
GameLogic
- round : int = 1
- cardsToDealPerRound : int[] = { 1 , 2 , 3 , 4 , 5 , 6 , 5 , 4 , 3 , 2 , 1 }
+ GameLogic()
+ getRound() : int
+ setRound(round : int) : void
+ getTableHand() : TableHand
+ setTableHand(player : Player, playerCard : Card) : void
+ getScoreboard() : Scoreboard
+ getLeadSuitCard() : Card
+ setLeadSuitCard(leadSuitCard : Card) : void
+ setPlayerOrder(round : int) : void
+ initialisePlayers() : void
+ getArrayOfPlayers() : ArrayOfPlayers
+ getDealer() : int
+ setDealer(round : int) : void
+ clockWiseNext(id : int) : int
+ getDeck() : Deck
+ initialiseDeck() : void
+ getTrumpCard() : Card
+ setTrumpCard() : void
+ setPlayersHand(round : int) : void
+ startNewGame() : ArrayOfPlayers
+ startSubRound(roundNumber : int) : void
+ startRound() : void
ArrayOfPlayers
+ addPlayer(player : Player) : void
+ getPlayerByIndex(index : int) : Player
+ getNumberOfPlayers() : int
+ getArrayOfPlayers() : ArrayList<Player>
+ updatePlayerStates(players : ArrayList<Player>) : void
- arrayOfPlayers
- players
*
Player
- id : int
- position : int
- bid : int
- isTrickWinner : boolean
- isDealer : boolean
+ Player(id : int, position : int)
+ getPlayableHand(leadSuit : Suit, trumpSuit : Suit) : ArrayList<Card>
+ getPlayerId() : int
+ getPosition() : int
+ setPosition(position : int) : void
+ getHand() : PlayerHand
+ setHand(c : Card) : void
+ removeFromHand(cardIndex : int) : Card
+ removeFromHand(card : Card) : Card
+ getBid() : int
+ setBid(bid : int) : void
+ getTrickWinner() : boolean
+ setTrickWinner(value : boolean) : void
+ isDealer() : boolean
+ setIsDealer(value : boolean) : void
+ toString() : String
+ getAvailableBids(tricksToWinForRound : int, totalBidsSoFar : int) : ArrayList<Integer>
Computer
+ Computer(id : int, position : int)
- percentOfTrumpAndHigher(myHand : PlayerHand, trumpSuit : Suit) : double
- indexOfBid(percentOfTrumpAndHigher : double, numPossibleBids : int, medianIndex : int) : int
+ bidWinningTricks(totTricksInRound : int, sumOfBidsInTrick : int, trumpSuit : Suit) : void
+ playCard(trumpSuit : Suit, leadSuit : Suit, highestPlayedCard : Card) : Card
SelectCard
+ SelectCard()
+ pickCardFirstPlayerPositive(playerHand : PlayerHand, trumpSuit : Suit) : Card
+ pickCardFirstPlayerZero(playerHand : PlayerHand, trumpSuit : Suit) : Card
+ pickCardNextPlayerPositive(playerHand : PlayerHand, trumpSuit : Suit, leadSuit : Suit, highestPlayedCard : Card) : Card
+ pickCardNextPlayerZero(playerHand : PlayerHand, trumpSuit : Suit, leadSuit : Suit, highestPlayedCard : Card) : Card
- hand
- leadSuitCard
- trumpSuitCard
Card
- sortRankMajorOrder : boolean = false
+ Card(suit : Suit, rank : Rank, cardFace : ImageIcon)
+ getFilename(suit : Suit, rank : Rank) : String
+ getSuit() : Suit
+ getRank() : Rank
+ getCardImage() : ImageIcon
+ toString() : String
+ rankToString() : String
+ suitToString() : String
+ setRankMajorSort() : void
+ setSuitMajorSort() : void
+ compareTo(otherCardObject : Object) : int
+ isSameAs(card : Card) : boolean
- suitValue
- hand
{ordered}
Hand
hand : List = new ArrayList()
+ addCard(card : Card) : void
+ getCard(index : int) : Card
+ removeCard(card : Card) : Card
+ removeCard(index : int) : Card
+ discardHand() : void
+ getNumberOfCards() : int
+ sort() : void
+ isEmpty() : boolean
+ containsCard(card : Card) : boolean
+ findCard(card : Card) : int
+ compareTo(otherHandObject : Object) : int
+ evaluateHand() : int
+ toString() : String
+ replaceCard(oldCard : Card, replacementCard : Card) : boolean
PlayerHand
+ PlayerHand()
+ getPlayableHand(leadSuit : Suit, trumpSuit : Suit) : ArrayList<Card>
+ evaluateHand() : int
+ getHand() : ArrayList<Card>
+ toString() : String
BidZeroNextCardComparator
+ BidZeroNextCardComparator(trumpSuit : Suit, leadSuit : Suit)
+ compare(c1 : Card, c2 : Card) : int
BidPosFirstCardComparator
+ BidPosFirstCardComparator(trumpSuit : Suit)
+ compare(c1 : Card, c2 : Card) : int
BidZeroFirstCardComparator
+ BidZeroFirstCardComparator(trumpSuit : Suit)
+ compare(c1 : Card, c2 : Card) : int
- trumpSuit
- leadSuit
- trumpSuit
E
<<interface>>
Comparator
<<bind>>
<E->Card>
<<interface>>
Comparator<Card>
Suit
- name : String
- symbol : String
+ VALUES : List = Collection...
- Suit(nameValue : String, symbolValue : String)
+ getName() : String
+ getSymbol() : String
+ toString() : String
+ compareTo(otherSuitObject : Object) : int
+ SPADES
+ CLUBS
+ DIAMONDS
+ HEARTS
trumpSuit
ArrayList<Integer>
E
ArrayList
<<bind>>
<E->Integer>
<<bind>>
<E->Player>
ArrayList<Player>

Appendix

Completed Game Images




The trump suit is: Diamonds
The lead suit is: Clubs
Round: 1
Subround: 1 / 1
Position: 2
Tricks to win: 0
Tricks won: 0
Robot B
0
Robot A
0
Position: 1
Tricks to win: 0
Tricks won: 0
Robot C
0
Position: 3
Tricks to win: 0
Tricks won: 0
Your turn
Position: 4
Tricks to win: 0
Tricks won: 0
Player 1
0


The trump suit is: Spades
Round: 3
Subround: 2 / 2
Position: 4
Tricks to win: 0
Tricks won: 0
Robot B
20
Robot A
-1
Position: 3
Tricks to win: 1
Tricks won: 0
Robot C
0
Position: 1
Tricks to win: 0
Tricks won: 1
...
Position: 2
Tricks to win: 2
Tricks won: 0
Player 1
-22
BID WINDOW
Select one from the available bids:
3
2
1
0


The trump suit is: Spades
The lead suit is: Diamonds
Round: 3
Subround: 1 / 3
Position: 4
Tricks to win: 0
Tricks won: 0
Robot B
20
Robot A
-1
Position: 3
Tricks to win: 0
Tricks won: 0
...
Position: 2
Tricks to win: 2
Tricks won: 0
Player 1
-22
Message
Round: 3
Subround: 1
Winner is: Player 1
OK


Estimation Game
The trump suit is: Clubs
The lead suit is: Diamonds
Round: 5
Subround: 1 / 5
Tricks to win: 0
Tricks won: 0
Robot C
0
Robot B
-46
Tricks to win: 1
Tricks won: 0
Robot D
-4
Tricks to win: 3
Tricks won: 0
Your turn
Tricks to win: 3
Tricks won: 0
You
-3