Faculty of Engineering

MİŞTİ GAME (Project)

SE115, Introduction To Programming II, 2023-2

Izmir University of Economics - Faculty of Engineering, Fevzi Çakmak, Sakarya Cd. No:156, 35330

Balçova/İzmir.

ALİ AHMET ERDOĞDU

BIOMEDICAL & SOFTWARE ENGINEERING

20200612009

AYÇA DERNEK

ELECTRIC AND ELECTRICAL & COMPUTER ENGINEERING

20200607014

GİZEM GÜLLÜ

SOFTWARE ENGINEERING

20210601028

13. May 2023

Contents

Contents 2
Main Class 3
Deck Class 11
Card Class 12
Winner Class 13
Player Class (Abstract) 13
BothPlayer Class (Abstract) 14
HumanPlayer Class 14
NovicePlayer Class 15
RegularPlayer Class 15
ExpertPlayer Class 15
Parameter Exception Class 18
GamerCategory Exception Class 18
The UML Class Diagram 19
SAMPLE RUN 21

Main Class

The game is played in the “Main” class. First of all, the “static” values are defined.

The values are ;

- ArrayList<Card> hand1 , hand2, hand3 , hand4 ;
    - These represent cards gamers hand during the game.
- ArrayList<Card> storeCards1, storeCard2, storeCards3 , storeCards4
    - These represent to cards gamers get when make “MİŞTİ” or card faces and the cards’ faces on the board are the same
- static int playerCount = 0;
    - The value represents the number of players. The game is played according to this value.
- static int humanCounter = 0;
    - According to the rule, there cannot be more than one player. So, the value allows controlling the number of “HUMAN” players.
- static String firstGamerName, secondGamerName, thirdGamerName, forthGamerName;
    - The value represents players' names.
- static String firstGamerCategory, secondGamerCategory, thirdGamerCategory, forthGamerCategory;
    - The value represents players' levels.
    - “HUMAN” “EXPERT-BOTH” “NOVICE-PLAYER” “REGULAR-BOTH
- static Player _firstGamer, _secondGamer, _thirdGamer, _fortGamer;
    - The values represent gamers.

Before starting, game parameters are taken from the command line. These are player count, the name of the point file, round count, verbose mode, gamers’ names, and gamers’ levels.

The parameters are controlled by the methods described in the following paragraph.

1. **public static void getPlayerCount(String args)**

The method checks the player count. “static int playerCount = 0; ”. This is a static value so that it can be reachable from everywhere in the class. Because the value comes from String[] args, the method takes the String value. To prevent any errors, try-catch blocks prevent our codes. If players count more than 4 and less than 2, Exceptions are thrown then parameters exception is thrown. The parameters exception is shown when there are any errors, it shows an error message about parameters.

2. **public static void checkFile (String args)**

The method checks whether the file is provided from the command line as the point file is available. If it is not available, File Not Found exceptions are thrown then the Parameters Exception is thrown. The parameters exception is shown when there are any errors, it shows an error message about parameters.

3. **public static boolean takeVerboseValue(String args)**

The method checks whether the player wants to play in verbose mode. If the string value of “true” or “false” is not entered a Parameter Exception is thrown. On the other hand, if the string value of “true” is entered the method returns true else returns false.

4. **public static void getRoundCount (String args)**

The method checks the round count. “static int roundCount = 0; ”. This is a static value so that it can be reachable from everywhere in the class. Because the value comes from String[] args, the method takes the String value. the value set how many times the game is played. To prevent any errors, try-catch blocks prevent our codes. When any incomplete or wrong parameters are entered, Exceptions are thrown. On the other hand, Parameters Exception is thrown as well. The parameters exception is shown when there are any errors, it shows an error message about parameters.

5. **public static void checkPlayerCategory (String gamerCategory)**

The gamers' name and their levels are taken from the command line. The method checks whether the gamers’ names are provided correctly. If the category is not provided as one of the levels, the GamerCategory exception is thrown. Also, if there is “HUMAN” category more than one; the same exception is thrown.

When all parameters are controlled and ensured the correction of parameters, the next step is gamers and their references are assigned according to parameters are given.

“”””

```
String[] gamersName = {firstGamerName, secondGamerName, thirdGamerName, forthGamerName};

String[] gamersCategories = {firstGamerCategory, secondGamerCategory, thirdGamerCategory,
    forthGamerCategory};

 _gamers = new Player[]{_firstGamer, _secondGamer, _thirdGamer, _fortGamer};

 ArrayList<ArrayList<Card>> hands = new ArrayList<ArrayList<Card>>(4);

        hands.add(hand1);
        hands.add(hand2);
        hands.add(hand3);
        hands.add(hand4);

 ArrayList<ArrayList<Card>> storeCards = new ArrayList<ArrayList<Card>>(4);

        storeCards.add(storeCards1);
        storeCards.add(storeCards2);
        storeCards.add(storeCards3);
        storeCards.add(storeCards4);
```

“””

In the code block, the gamers' names and levels are assigned to arrays, and hands and storedCards are added to Arraylists. After this step, in a while loop gamers are created and added to _gamers arrays depending on given parameters.

The loop first checks if the correct number of player arguments were passed and prints an error message if not. Within the loop, it extracts the player's name and category and checks if the category is valid. Based on the category, it creates an instance of the appropriate player type and adds it to an array of players. The loop continues until it has created instances for all specified players. If an ArrayIndexOutOfBoundsException occurs, it catches the exception and throws a ParametersError. The code is shown following this paragraph

“”””

```
int i = 0;

if ((args.length - 4) / 2 < playerCount || (args.length - 4) / 2 > playerCount) {
  System.err.println("\nYour players data were not given correctly \nThere are " + playerCount + " players but " + ((args.length - 4) / 2) + " players were provided");
  try {
     throw new ParametersError();
  } catch (ParametersError ex) {
     System.err.println(ex.getMessage());
     System.exit(1);
    }
}

while (true) {
  try {

     gamersName[i / 2] = args[i + 4]; // taking players name

     gamersCategories[i / 2] = args[i + 5]; //

     if (gamersCategories[i / 2].equals("HUMAN")) {
        humanCounter++;
     }
     checkPlayerCategory(gamersCategories[i / 2]);
     switch (gamersCategories[i / 2]) {
        case "HUMAN" ->
              _gamers[i / 2] = new HumanPlayer(gamersName[i / 2], hands.get(i / 2), 0, "HUMAN", storeCards.get(i / 2));
        case "EXPERT-BOTH" ->
              _gamers[i / 2] = new ExpertPlayer(gamersName[i / 2], hands.get(i / 2), 0, "EXPERT-BOTH", storeCards.get(i / 2));
        case "REGULAR-BOTH" ->
              _gamers[i / 2] = new RegularPlayer(gamersName[i / 2], hands.get(i / 2), 0, "REGULAR-BOTH", storeCards.get(i / 2));
        case "NOVICE-BOTH" ->
              _gamers[i / 2] = new NovicePlayer(gamersName[i / 2], hands.get(i / 2), 0, "NOVICE-BOTH", storeCards.get(i / 2));
     }
     i += 2;
     if ((i / 2) == playerCount)
        break; //
  } catch (ArrayIndexOutOfBoundsException e) {
     try {

        throw new ParametersError();

     } catch (ParametersError ex) {
          System.err.println(ex.getMessage());
          System.exit(1);
                }
          }
  }
```

“”””

STARTING OF GAME

The codes of the main game are in the main class and are located in four loops. The first three of these cycles control the number of rounds played, whether there are cards in the deck, and whether the players have cards in their hands, respectively. The last loop travels between the players, allowing each player to play the game.

1.While Loop: This loop controls the number of rounds of the game played and continues as long as the number of rounds is less than or equal to the number that the player originally wrote to the terminal. When this loop is run, first a message that the game has started is printed on the screen, and then the variables required for the game are created.

Deck: It is the variable where the cards used while playing the game are kept. At the beginning of each “round”, it is first formed as a fifty-two card deck, which is then shuffled and cut in order.

Board: The variable in the game where the cards are discarded. It is created at the beginning of each "round" and is automatically dealt the first four cards in the deck. The discarded card during the game is added to this variable.

Throwed: It is a static variable that we created in the "ExpertPlayer" class. When cards are dealt to the board, the same cards are added to this variant at the beginning of the game. During the game, each card that the players throw is added to this list.

Topcard: It is the variable created in the "both player" class. At the beginning of the game, the card in the third index of the board variable is assigned to this variable. this keeps the top card on the board. This card is updated during the game. If there is no card on the board, this variable is stored as “null”.

Hand_number: This variable is used to indicate which hand the information printed for verbose mode belongs to. It is increased during each hand during the game.

Verboselist: In this variable, for the "verbose mode", the cards in each hand of the players at the beginning of each hand, their scores and the cards discarded in that hand are stored as "stringing" according to the order in which they are discarded. At the end of the game, if "verbose mode" is "true", they are printed on the screen.

Lastgamer: This variable holds the player who took the last card in the game. At the end of the game, the cards remaining on the board are given to the person who took the last card during the game.

After the codes in the other loops in this loop are executed, three basic operations are performed before proceeding to the next round.

• During the game, the cards taken by the players are navigated with the help of the for loop and printed on the screen as a list.

• In order to determine the winning player, a new list is created and copied here with the help of the for loop. Then it is visited among these players and it is checked whether the player's score is greater than the next player's score. If it is large, a new variable is created and this

variable is referenced by this player. Players are swapped. This process repeats for all players on the list. Then, with the help of another for loop, the scores of the players are printed on the screen in order. Then, the first player is given as a parameter to the "settopten" function to update the top ten list after pressing the screen.

• Before moving on to the next round, players are cycled through the for loop and their scores and cards are reset. After the "throwed" card list is reset, the number of rounds is increased by one, and the next round is passed.

2. While Loop: The codes in this loop are executed at the beginning of each hand. At the beginning of each hand, players are dealt cards one by one using the "dealcard" function. Then the hand number for verbose made and the starting scores of each player and the cards in their hand are recorded with the help of the for loop. The "turn number" variable is created to facilitate the recording of cards discarded in each round for the hand to be played. At the end of each hand, two main functions are performed.

• The scores of the cards remaining on the board at the end of the hand are added to the score of the last person to have received a card in the game. For this, the cards are visited with the for loop and the scores are added to the player one by one. At the end of the loop, these cards are added to the list of cards won by the player, and the board is printed for the last time.

• If the verbose mode is “true”, the strings in the “verbose list” are navigated with the help of a loop and these strings are printed on the screen one after the other.

3. While Loop: In this section, it is checked whether the players have cards in their hands. As long as the players have cards, the players continue to take turns throwing cards. In this section, a "cardlist" variable is created to be printed in verbose mode, which keeps the cards thrown in order for each round. It is added to the "verbose list" in which round the game is. At the end of this section, the "cardlist" variable is converted to a string and added to the "verbose list" and the elements in the "card list" are deleted.

For loop: In this part of the game, the players throw cards respectively with the for loop, and the discarded cards are evaluated and necessary actions are taken. At the beginning of the level, the board is shown to each player, and then the card-discarding functions are called, allowing players to discard cards. The discarded card is referenced to the "throwcard" variable. And it is added to the variable “card list” for verbose mode. After these operations are done, the discarded card begins to be evaluated. The evaluation part consists of 2 main parts and the second part consists of 3 sub-parts.

Part 1: This part of the code works if there are no cards on the board. For this, the "topcard" variable is checked. And if this variable is empty this block is executed. Since there are no cards on the board, it is impossible for the player to pick up the cards on the board and to make”mişti”. Therefore, the player's discarded card is printed on the board and the card is added to the board, "topcard" and "throwed" variables, respectively. Finally, the card is deleted from the list where the cards in the player's hand are kept.

Part 2: This part is run if there is at least one card on the board and the card is evaluated in 3 main ways.

1. In this part, it is checked whether the card at the top of the board is the same as the discarded card. If the value of the discarded card and the top card are the same, the codes in this section are executed. In this section, the card is examined in 2 main cases. These are, respectively, the case of having one card on the board (“MİŞTİ”) and the case of having more than one card on the board (the case where all the cards on the board are won).

   1.1. If there is a card on the board and this card and the discarded card match, this code block is executed. The card that the player discarded is written on the board and it is stated that the player has made “mişti”. Then, five times the sum of the points of the discarded card and the cards on the board are added to the player's score. The card discarded by the player is added to the "throwed" list and the "topcard" variable is updated to "null" as there are no cards left on the board. The discarded card is added to the board for convenience and all cards on the board are added to the user's owned cards list. Finally, the board is cleared, the card is cleared from the player's hand, and the variable that holds the last card winner is updated.

   1.2. In this part, there is more than one card on the board and therefore it is not possible to make “mişti”. Instead, the player takes all the cards on the board. Unlike the previous part, a text is written on the terminal that the player has taken all the cards on the board, the score of the discarded card is added to the player and the scores of the cards on the board are added to the player's score with the help of the "for" loop. The rest of the codes are the same as in the previous condition.

2. In this part, the case of the discarded card being a jack is examined. If the value of the discarded card is equal to the jack card, this card takes all the cards on the board thanks to its feature. The code design is the same as if there is more than one card on the board condition in the upper part.
3. In this section, the case of having a card on the board but the discarded card does not match the top card and is not equal to the jack card is examined. In this case, which card the player throws is printed on the console. The discarded card is then added to the board and to the "throwed" list. The variable "topcard" is mapped to this card and the card is removed from the player's hand. Since the player does not win a card, his score does not change.

ENDING OF GAME

6. **public static void showBoard (ArrayList < Card > board)**

The method shows the cards on the board when it is called. It takes a parameter as Arraylist< Card >. So the ArrayList stores the cards players played on board.

7. **public static void setTopTen (Player winner)**

.
The method takes a "Player" object as input, which represents the winner of a game. It tries to open the "Score.txt" file using the "FileReader" and "BufferedReader" objects. If the "Score.txt" file does not exist, it creates the file and initializes it with 10 empty score entries.t then reads the existing score entries in the file and stores them in an array of "Player" objects called "winners". It compares the score of the input "Player" object to the scores in the "winners" array to determine if the input "Player" has a higher score than any of the existing scores. If the input "Player" has a higher score than any of the existing scores, it replaces the score of the lowest-ranked "Player" in the "winners" array with the input "Player". It then sorts the "winners" array in descending order by score. Finally, it writes the sorted "winners" array to the "Score.txt" file using a "FileWriter" object.

Deck Class

The Deck class represents a deck of cards and provides methods for managing and manipulating the deck.

1. **Deck(File pointFile):** This is the constructor of the Deck class. It takes a File object pointFile as a parameter, representing the file containing the card data. The constructor initializes the pointFile instance variable and creates an empty ArrayList called cards to store the cards. It then calls the readCardsFromFile() method to read the card data from the file and populate the cards list. This design allows flexibility in using different card data files and separates the responsibility of card reading from the constructor.

2. **readCardsFromFile():** This method reads the card data from the file specified in the pointFile instance variable. It creates a new ArrayList called cards to store the cards. The method attempts to open the file using a Scanner object and iterates through each line of the file. For each line, it splits the line into individual attributes using the comma as a delimiter. It then creates a Card object with the extracted attributes (suit, card face, and points) and adds it to the cards list. Finally, it closes the Scanner and returns the populated cards list. This design separates the card reading logic into a dedicated method, promoting code modularity and enhancing readability.

3. **shuffle():** This method shuffles the cards in the deck. It uses the Collections. shuffle() method, which employs a random shuffling algorithm to reorder the elements in the cards list. This design choice leverages the robustness and efficiency of the Collections. shuffle() method, ensuring a fair and randomized shuffling of the cards.

4. **cut():** This method performs a cut operation on the deck, simulating a common card manipulation technique. It generates a random cutIndex within the range of the number of cards in the deck. The method then creates a new ArrayList called cuttedCards to store the second half of the deck. It uses the subList() method to obtain a sublist starting from cutIndex to the end of the cards list, effectively extracting the second half of the deck. It clears the elements from the original cards list using subList().clear(). Finally, it adds the cuttedCards sublist to the beginning of the cards list using addAll(). This design choice enables the simulation of a cut operation commonly performed in card games, enhancing the realism and gameplay possibilities.

5. **deal(int numCards):** This method deals with a specified number of cards from the deck. It takes an integer numCards as a parameter, representing the number of cards to be dealt. The method creates an empty ArrayList called dealtCards to store the dealt cards. It uses a for loop to iterate numCards times. In each iteration, it removes the first card from the cards list using remove(0) and adds it to the dealtCards list using add(). If the deck does not have enough cards, the loop terminates early. Finally, it returns the dealtCards list. This design provides a convenient way to distribute cards to players or any other game-related components.

As for the challenges encountered during the implementation, the main difficulty was handling file input and parsing the card data. The readCardsFromFile() method had to handle potential exceptions, such as FileNotFoundException, when attempting to open the card file. To address this, appropriate exception handling using a try-catch block was implemented to print the stack trace for easier debugging and error identification.
Another challenge was parsing the card data from each line of the file. The code relied on a specific format where the card attributes were separated by commas. However, if the file format deviated from this structure or contained unexpected data, it could potentially cause errors during parsing. To mitigate this, the code assumed a well-formed file format and relied on the consistency of the data. However, in a more robust implementation, additional error handling and validation could be added to handle potential data inconsistencies.

In summary, the Deck class encapsulates the functionality of a deck of cards and provides methods for initializing, shuffling, cutting, and dealing cards. The design promotes code modularity, separation of concerns, and flexibility in adapting the class to different card games or scenarios.

Card Class

The Card class represents a playing card and encapsulates its attributes such as suit, card face, and points. It provides methods for setting and retrieving these attributes, as well as a toString() method for obtaining a string representation of the card.
The class contains the following instance variables:

- suit: A private string variable that stores the suit of the card, such as "Spades" or "Hearts".
- cardFace: A private string variable that represents the face value of the card, such as "Ace", "King", or "Queen".
- points: An integer variable that holds the point value associated with the card.

The class provides two constructors:

- Card(String suit, String cardFace, int points): A constructor that takes in the suit, card face, and points as parameters and initializes the corresponding instance variables.
- Card(): A default constructor that sets the suit and card face to null and the points to zero.

The class also includes getter and setter methods for each instance variable, allowing external classes to access and modify the card's attributes as needed.
Additionally, the class overrides the toString() method, which returns a string representation of the card by concatenating its card face and suit.

Winner Class

The Winners class extends the Player class, indicating that it is a specialized type of player in a game. It represents a player who has achieved the status of being a winner. The Winners class inherits all the attributes and methods from the Player class, allowing it to access and modify player-specific information.
The class includes a constructor that takes in several parameters: name, hand, score, level, and storedCards. These parameters are passed to the superclass (Player) constructor using the super keyword, which initializes the corresponding attributes inherited from the Player class.
By extending the Player class, the Winners class inherits the behavior and characteristics of a player, including the ability to have a name, a hand of cards, a score, a level, and a collection of stored cards. It can also access the getter and setter methods defined in the Player class to retrieve and modify these attributes.

Player Class (Abstract)

It was created for players who will play the game. Functions and values that must be common and mandatory for all players are defined within this “class” structure. The reason why it was chosen as "Abstract" is to both reduce code duplication by using inheritance and not need to produce an object directly from this "class" structure.

First of all, these variables are defined since all of the players who will play the game, regardless of whether they are bots or humans, will have a name, a list of cards to keep the cards dealt to them during the game, and a score. Among these variables, the “name” variable is defined as a String, the “hand” variable as a Card array, and the “score” variable as Integer. The variable “level”, which is a String object, is to be used to define the types or levels of the players (for bot users) in the later coding stages of the game; The "storedcard" variable, which is a card array, has been added to this class to keep the cards won by the players during the game. All of the variables are defined as private to restrict access. Necessary constracter, “getter” and “setter” methods are also defined in this class.

Since all players will have to throw cards during the game, a common function structure called “playcard” has been added to this class structure so that it is overwritten by other classes that inherit this class.

BothPlayer Class (Abstract)

This class has been created as abstract so that there is no need to produce an object directly from its structure. This class was created because of the differences between both classes and the humanplayer class in the design process. A card object called “topcard” is created. It has been created as static in order to access and update this object more easily than other parts of the game, and to give the same “topcard” object to “regular” and “expert” bots throughout the game.

HumanPlayer Class

It has been created for players who will play the game by giving commands over the terminal. Apart from the "playcard" function, which is inherited from the "parent" class, it has one unique method called "showhand".

1. **showHand():** This function is created to show the user the cards in their hand. The reason why it is not in other player classes is because other players are bots. The function takes no parameters and navigates the player's cards with the help of a for loop. Prints each card side by side with a space between them

2. **placard ():** This function primarily prints the player's hand on the console with the help of the “showhand” function. Then, a text is printed to enter the sequence number of the card to be thrown to the player and the player is asked for the number. First of all, the received information is tried to be converted to an "integer". The value received in this block is expected to be equal to or less than the number of cards in the player's hand and at the same time equal to one of the numbers 1, or 2,3,4. If this equality is achieved, the index number will be equal to this number, and at the end of the function, one will be subtracted from the index number, and the card in this index will be returned. If not provided, the index number remains as -2 at the beginning of the code. In cases where the index number is -2 or the received value cannot be translated as "integer", the code will throw an error and this error will be caught by the catch block. In this block, a warning message is suppressed that the user has entered an incorrect value. And the cards in his hand are shown to the user again. Since the code inside the function (including the try, catch block) is written inside an endless while loop, this input process continues until the player enters the correct value.

NovicePlayer Class

This class structure is the simplest of bot users that will be played automatically by the computer. It contains the "playcard" function, which it inherits only by inheritance.

1. **playCard():** This function first sets the index number to 0, then checks whether the number of cards in the player's hand is greater than one. If it is greater than one, it creates a number between zero and the length of the list (not including the number where the list length is equal) with the help of the random library and assigns that number to the index number. At the end of the code, it returns the card in this index.

RegularPlayer Class

The player in this class is playing cards with a better strategy than the “novice” player and worse than the “expert” player. It has no additional functionality other than the "playcard" method, which is inherited.

1. **playCard():** This player's strategy for discarding is to discard a card that matches the top card of the deck. If not, discard the card with the lowest point value. For this, first, the cards in the player's hand are visited with the for loop. If there is an identical card to the card on the top of the deck, this card is returned by the function as a discarded card. Otherwise, a new "ArrayList" is created and the points of the cards in the player's hand are added to this list in order. In this way, the cards in the player's hand and the scores of those cards are in the same index in both lists. Then, using the collection library, the index of the smallest score in the list where the scores are kept is found and the card in this index is returned by the function as the card to be discarded.

ExpertPlayer Class

This class was created to represent the player with the most complex strategy the computer has ever played. Unlike other player classes, the "Expert Player" class has a separate variable called "throwed" to collect discarded cards. It does not contain any additional functions other than the "playcard" function that it inherited.

Throwed: Created to keep cards that have been discarded in-game. This variable has been created as "static" since all ExpertPlayer class players who will take place as players during the game will need to use the same list and also this variable will need to be changed dynamically throughout the game. Therefore, there is no need to create "setter" and "getter" methods.

The strategy of the Expert Player: The Expert player first checks whether there are cards on the board during the discard. For this, it checks the "topcard" variable and checks whether there is a card whose value is equal to this card, if any. If he finds such a card, he discards this card directly. If there is no such card, it checks the discarded card list. If there are cards in this list, he looks to see if any cards of the same value as the cards in his hand have been discarded and discard the card with the lowest score among the cards with the highest number of repetitions. The reason for this is that the card in his hand will not be the same as the one on the board and therefore he cannot take the cards on the board, so if his opponent takes the cards, the opponent's score will increase less. If there are no cards on the board and there are no cards on the discard list, the card in the hand with the lowest score discards.

1. **Playcard():** This function is divided into 4 main parts with an “else-if” structure. In the first part, if the player has only one card in his hand, that card is returned directly. In the second part, the card to be discarded is decided by considering the card on the board. In the third part, if there is no card on the board, the card to be discarded is decided by considering the previously discarded cards. In the fourth part, in cases where the situations in the first three parts cannot be used, only the points are taken into account and the card to be discarded is decided.

1. Part: In this part of the function, it is checked whether the number of cards in the player's hand is more than one. If the player has only one card in their hand, the code executes this block and returns the card at the zero "index" directly.

2. Part: In this section, “topcard” is checked and it is determined whether there is a card on the board. If the "topcard" variable is not "null", that is, it has a reference to a card object, this code block is executed. In this code block, first of all, the cards in the player's hand are checked to see if there is a match with the "topcard", and if so, this card is returned. If not, the same code as part 3 and part 4 is executed.

3. Part: In this section, it is checked whether there is a card that has been thrown in the game before. If any, a list named "repeat" is created that holds "integer" values. The cards in the player's hand are compared with the discarded cards, using the "for" loop. If the value of both cards is the same, one is added to the value in the same index as the index of this card in the "repeat" list.

At the end of the loop, with the help of the "collection" library, the number of times the most repeated card repeats is assigned to a variable called "cardrepeat". Then the point value of one of the cards with the highest number of repetitions is assigned to the variable “minpoint”. With the help of a "for" loop, the cards in the player's hand are reviewed. The card with the lowest score, whose number of repetitions is equal to the number of "cardrepeats", is found and that card is returned by the function. If there is no card in the "throwed" list, the same code as in part 4 is executed.

4. Part: This part is executed when there are no cards on the board and at the same time the “throwed” card list is empty. In this part, a new "arraylist" is created and the points of the cards in the player's hand are added to this list in order. In this way, the cards in the player's hand and the scores of those cards are in the same index in both lists. Then, using the collection library, the index of the smallest score in the list where the scores are kept is found and the card in this index is returned by the function as the card to be discarded.

Parameter Exception Class

In this class, when game parameters are provided wrong; it is thrown.

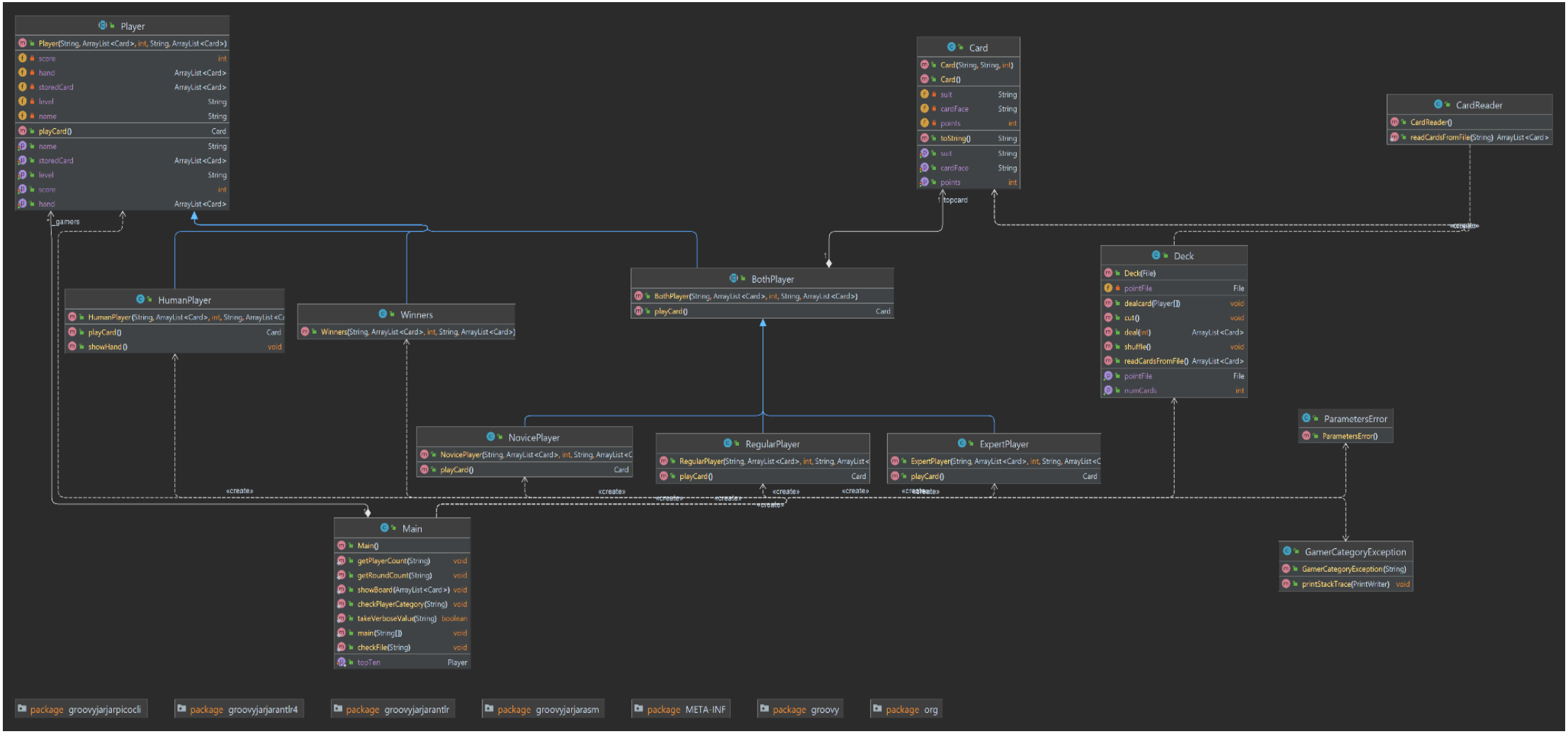
```
public class ParametersError extends Exception{
   public ParametersError(){
       super("\nPlease provide parameters correctly \n" +
               "1 - Player Count \n" +
               "2 - Point file name \n" +
               "3 - Round Count \n" +
               "4 - Verbose mode \n" +
               "***** Provide player data as player count as.
These are just examples *****\n"+
               "***** !!! Please provide player names' combined
for example not 'ALİ AHMET' provided as !!! 'ALİAHMET' ******\n" +
               "***** Gamer categories are  'HUMAN' 'NOVICE-BOTH'
'EXPERT-BOTH' 'REGULAR-BOTH' \n" +
               "5 - First player name \n" +
               "6 - First player level  \n" +
               "7 - Second player name  \n" +
               "8 - Second player level  \n" +
               "9 - Third player name  \n" +
               "10 - Third player level  \n" +
               "11 - Forth player name  \n" +
               "12 - Forth player level  \n");
   }
}
```

GamerCategory Exception Class

In this class, when gamers' categories are provided wrong; it is thrown.

```
public class GamerCategoryException extends Exception{

   public GamerCategoryException(){
      super("Please enter category correctly");
   }
   public GamerCategoryException(String message){
       super(message);
   }
```

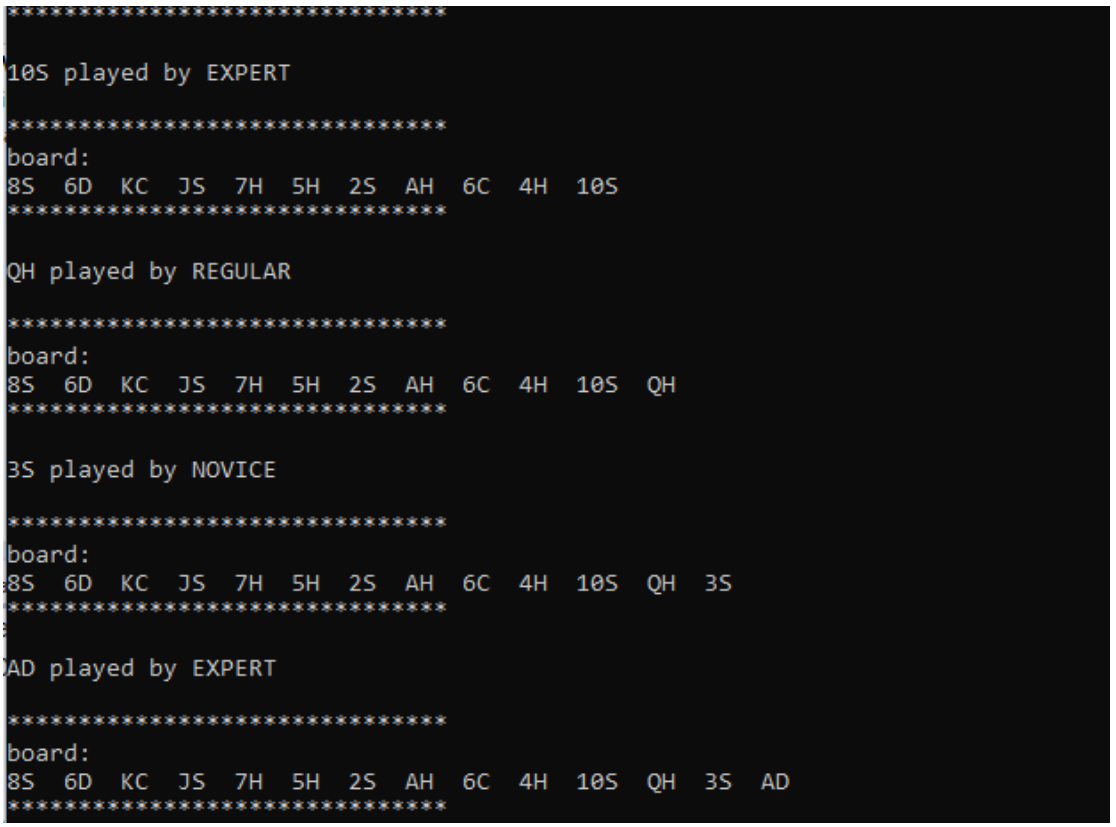
The UML Class Diagram

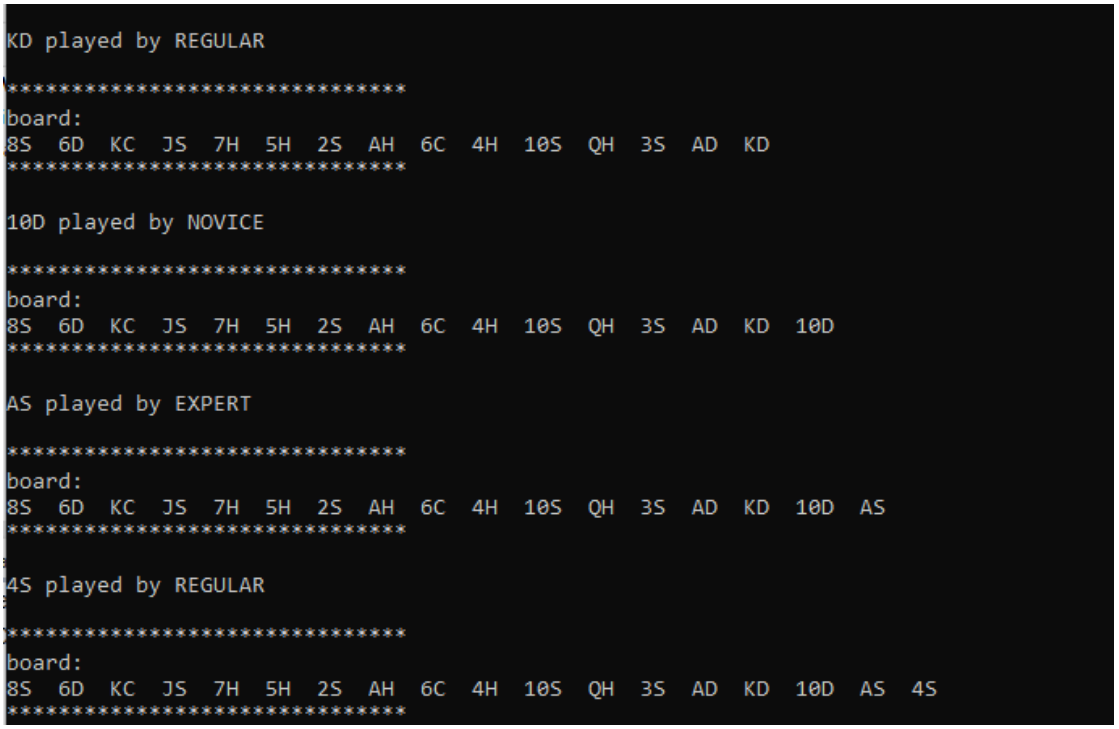


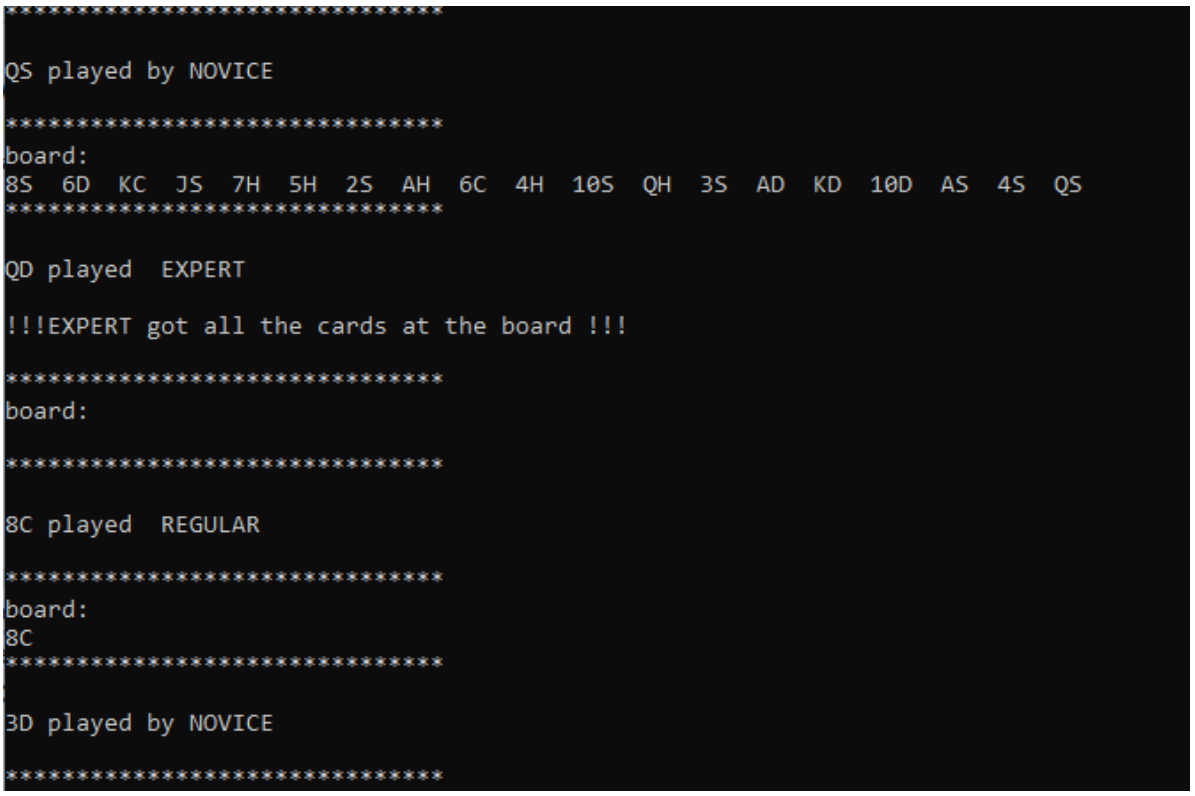
SAMPLE RUN

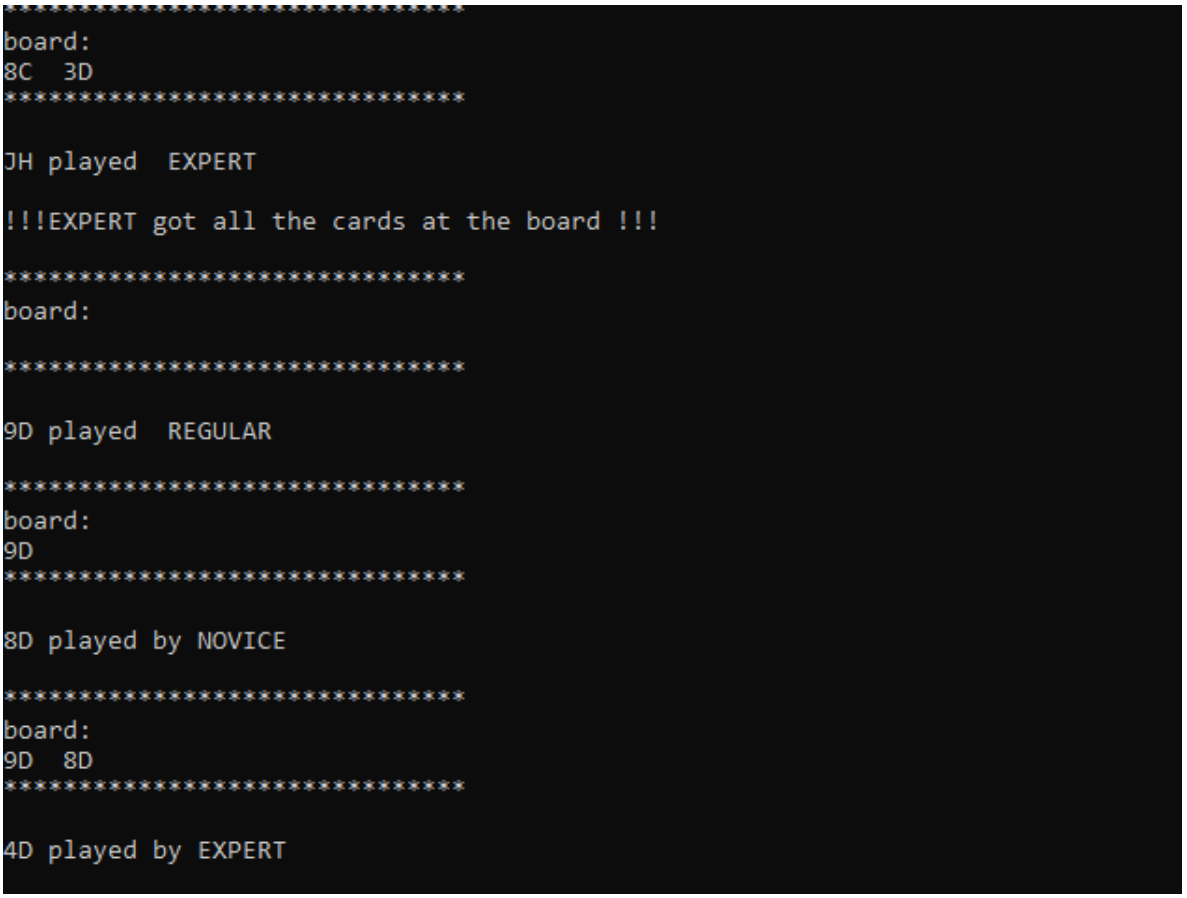
```
C:\Users\HP\Desktop>java -jar MISTIGAME.jar 3 Cards 1 True EXPERT EXPERT-BOTH REGULAR REGULAR-BOTH NOVICE NOVICE-BOTH
Our Gamer are :
1 EXPERT EXPERT-BOTH
2 REGULAR REGULAR-BOTH
3 NOVICE NOVICE-BOTH
-- Welcome to Mişti Game 1th round --
*******************************
board:
8S  6D  KC  JS
*******************************

7H played by EXPERT

*******************************
board:
8S  6D  KC  JS  7H
*******************************

5H played by REGULAR

*******************************
board:
8S  6D  KC  JS  7H  5H
*******************************
```

```
*******************************

2S played by NOVICE

*******************************
board:
8S  6D  KC  JS  7H  5H  2S
*******************************

AH played by EXPERT

*******************************
board:
8S  6D  KC  JS  7H  5H  2S  AH
*******************************

6C played by REGULAR

*******************************
board:
8S  6D  KC  JS  7H  5H  2S  AH  6C
*******************************

4H played by NOVICE

*******************************
board:
8S  6D  KC  JS  7H  5H  2S  AH  6C  4H
*******************************
```


****
10S played by EXPERT
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S
****
QH played by REGULAR
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH
****
3S played by NOVICE
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S
****
AD played by EXPERT
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD
****

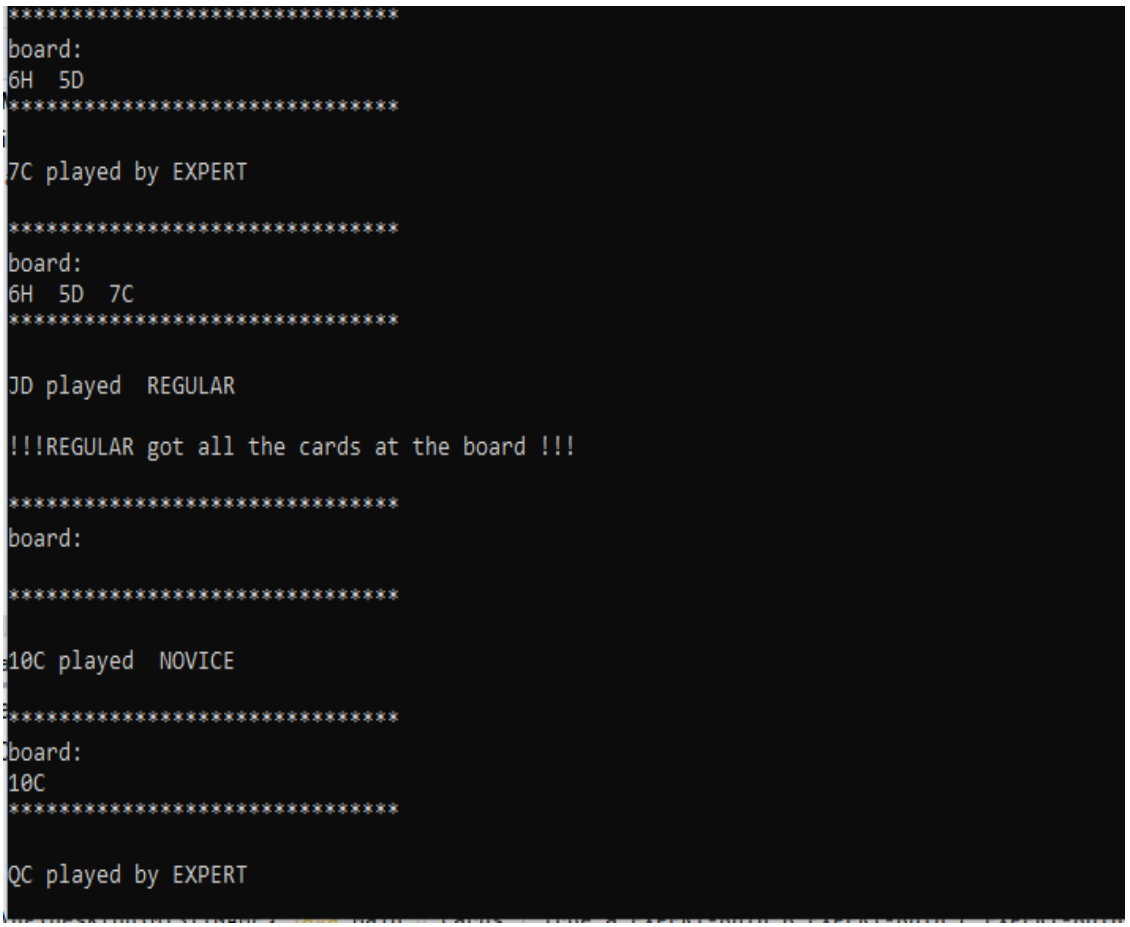

KD played by REGULAR
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD KD
****
10D played by NOVICE
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD KD 10D
****
AS played by EXPERT
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD KD 10D AS
****
4S played by REGULAR
****
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD KD 10D AS 4S
****

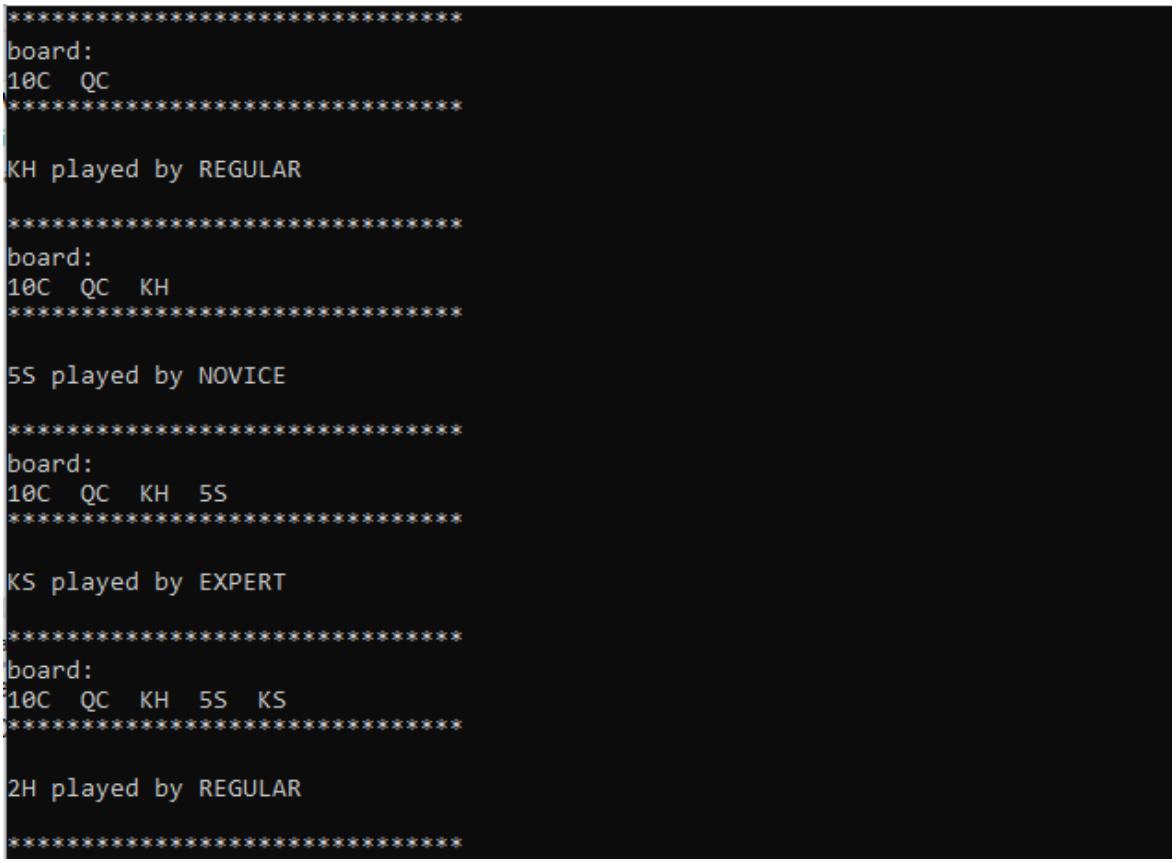

QS played by NOVICE
board:
8S 6D KC JS 7H 5H 2S AH 6C 4H 10S QH 3S AD KD 10D AS 4S QS
QD played EXPERT
!!!EXPERT got all the cards at the board !!!
board:
8C played REGULAR
board:
8C
3D played by NOVICE

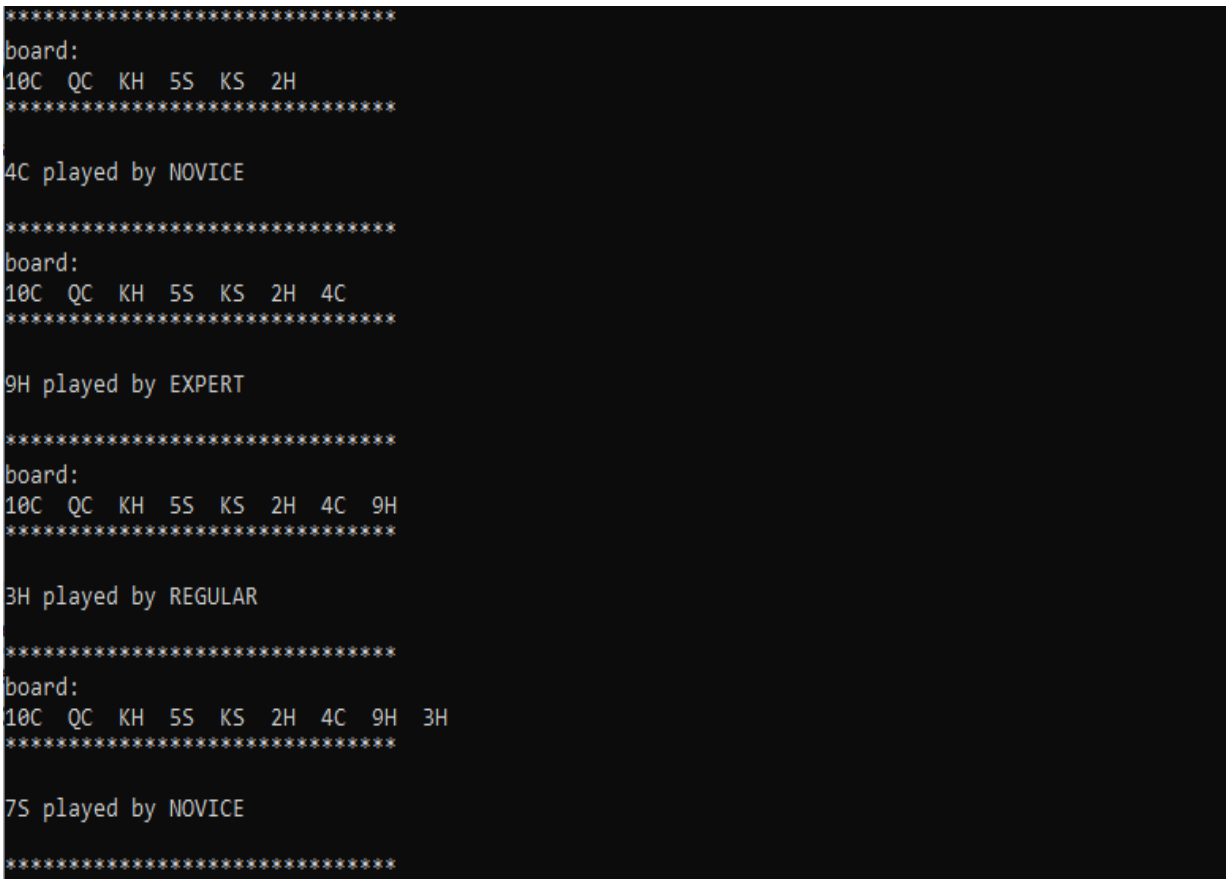

board:
8C 3D
JH played EXPERT
!!!EXPERT got all the cards at the board !!!
board:
9D played REGULAR
board:
9D
8D played by NOVICE
board:
9D 8D
4D played by EXPERT

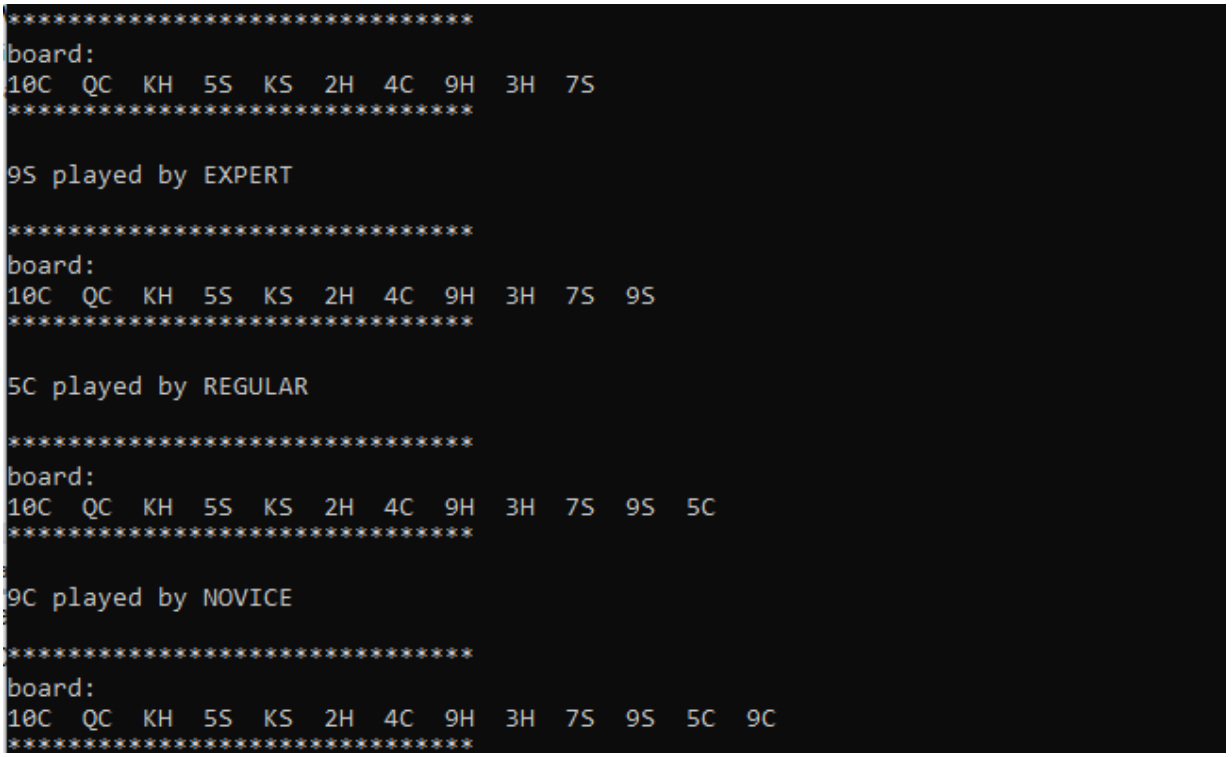
```
*******************************
board:
9D  8D  4D
*******************************

10H played by REGULAR

*******************************
board:
9D  8D  4D  10H
*******************************

JC played  NOVICE

!!!NOVICE got all the cards at the board !!!

*******************************
board:

*******************************

8H played  EXPERT

*******************************
board:
8H
*******************************

3C played by REGULAR
```

```
*********************************
board:
8H  3C
*********************************

2C played by NOVICE

*********************************
board:
8H  3C  2C
*********************************

2D played  EXPERT

!!!EXPERT got all the cards at the board !!!

*********************************
board:

*********************************

6H played  REGULAR

*********************************
board:
6H
*********************************

5D played by NOVICE
```


****
board:
6H 5D
****
7C played by EXPERT
****
board:
6H 5D 7C
****
JD played REGULAR
!!!REGULAR got all the cards at the board !!!
****
board:
****
10C played NOVICE
****
board:
10C
****
QC played by EXPERT


****
board:
10C QC
****
KH played by REGULAR
****
board:
10C QC KH
****
5S played by NOVICE
****
board:
10C QC KH 5S
****
KS played by EXPERT
****
board:
10C QC KH 5S KS
****
2H played by REGULAR
****


****
board:
10C QC KH 5S KS 2H
****
4C played by NOVICE
****
board:
10C QC KH 5S KS 2H 4C
****
9H played by EXPERT
****
board:
10C QC KH 5S KS 2H 4C 9H
****
3H played by REGULAR
****
board:
10C QC KH 5S KS 2H 4C 9H 3H
****
7S played by NOVICE
****


****
board:
10C QC KH 5S KS 2H 4C 9H 3H 7S
****
9S played by EXPERT
****
board:
10C QC KH 5S KS 2H 4C 9H 3H 7S 9S
****
5C played by REGULAR
****
board:
10C QC KH 5S KS 2H 4C 9H 3H 7S 9S 5C
****
9C played by NOVICE
****
board:
10C QC KH 5S KS 2H 4C 9H 3H 7S 9S 5C 9C
****

```
********************************

AC played by EXPERT

********************************
board:
10C  QC  KH  5S  KS  2H  4C  9H  3H  7S  9S  5C  9C  AC
********************************

6S played by REGULAR

********************************
board:
10C  QC  KH  5S  KS  2H  4C  9H  3H  7S  9S  5C  9C  AC  6S
********************************

7D played by NOVICE

********************************
board:
10C  QC  KH  5S  KS  2H  4C  9H  3H  7S  9S  5C  9C  AC  6S  7D
********************************
```

```
HAND 1
 EXPERT:  [AH, 10S, 7H, AD]  Score 0
 REGULAR:  [QH, 5H, KD, 6C]  Score 0
 NOVICE:  [4H, 3S, 10D, 2S]  Score 0


1.
 [7H, 5H, 2S]

2.
 [AH, 6C, 4H]

3.
 [10S, QH, 3S]

4.
 [AD, KD, 10D]
```

```
HAND 2
 EXPERT:  [AS, JH, 4D, QD]  Score 0
 REGULAR:  [4S, 8C, 10H, 9D]  Score 0
 NOVICE:  [3D, JC, 8D, QS]  Score 0


1.
 [AS, 4S, QS]

2.
 [QD, 8C, 3D]

3.
 [JH, 9D, 8D]

4.
 [4D, 10H, JC]
```

```
HAND 3
 EXPERT:  [8H, 7C, QC, 2D]  Score 179
 REGULAR:  [6H, JD, 3C, KH]  Score 0
 NOVICE:  [5D, 5S, 2C, 10C]  Score 41


1.
 [8H, 3C, 2C]

2.
 [2D, 6H, 5D]

3.
 [7C, JD, 10C]

4.
 [QC, KH, 5S]
```

```
HAND 4
 EXPERT:  [9H, 9S, AC, KS]  Score 194
 REGULAR:  [2H, 3H, 6S, 5C]  Score 28
 NOVICE:  [9C, 7S, 4C, 7D]  Score 41


1.
 [KS, 2H, 4C]

2.
 [9H, 3H, 7S]

3.
 [9S, 5C, 9C]

4.
 [AC, 6S, 7D]
```

```
In 1th
EXPERT [8S, 6D, KC, JS, 7H, 5H, 2S, AH, 6C, 4H, 10S, QH, 3S, AD, KD, 10D, AS, 4S, QS, QD, 8C, 3D, JH, 8H, 3C, 2C, 2D]
REGULAR [6H, 5D, 7C, JD, 10C, QC, KH, 5S, KS, 2H, 4C, 9H, 3H, 7S, 9S, 5C, 9C, AC, 6S, 7D]
NOVICE [9D, 8D, 4D, 10H, JC]
EXPERT   194
REGULAR   145
NOVICE   41
!!! congratulations  EXPERT !!!
EXPERT is winner 1th round
```