COMP41670: Software Engineering
2023 / 2024 Individual Project

Report

-Anant Singh (23200461)

```java
public class PatienceGame {

    public static final int Lanes = 7;
    public static final int Suits = Suit.values().length;
    public static final int Suitsize  = Rank.values().length;

    private Deck newdeck;
    private Stack<Card> pile;
    private List<Stack<Card>> newlanes;
    private List<Stack<Card>> newsuit;
    private boolean reveledcards;

    public PatienceGame() {
            // Constructor to initialize the game
        newdeck = new Deck();
        newdeck.shuffleDeck();
        pile = new Stack<>();
        newlanes = new ArrayList<>(Lanes);
        for (int i = 0; i < Lanes; i++) {
            newlanes.add(new Stack<>());
            for (int j = 0; j < i; j++) {
                newlanes.get(i).push(newdeck.pop());
            }
            Card card = newdeck.pop();
            card.CheckFaceUp();
            newlanes.get(i).push(card);
        }
        newsuit = new ArrayList<>(Suits);
        for (int i = 0; i < Suits; i++) {
            newsuit.add(new Stack<>());
        }
        reveledcards = false;
    }
    public boolean Possiblelityofmove() {
            // Check if there's a possibility of making a move in the game.
        return !newdeck.isEmpty() || !pile.isEmpty();
    }
    public void takingdraw() {
             // Draw a card from the deck
        if (!newdeck.isEmpty()) {
            Card card = newdeck.pop();
            card.CheckFaceUp();
            pile.push(card);
        } else {
            int numberOfCards = pile.size();
            for (int i = 0; i < numberOfCards; i++) {
                Card card = pile.pop();
                card.CheckFaceDown();
                newdeck.add(card);
            }
        }
    }
    public boolean moveIsPossible(Command command) {
            // Check if a given move is possible
        boolean isPossible = false;
        if (command.Moveingtopile() && command.MoveinginsideLane()) {
            if (!pile.isEmpty()) {
                Card card = pile.peek();
                Stack<Card> lane = newlanes.get(command.getIndex());
                if (lane.isEmpty() || (!lane.isEmpty() && lane.peek().ifNextInLane(card))) {
```

```
                isPossible = true;
            }
        }
    } else if (command.Moveingtopile() && command.MoveingToSuit()) {
        if (!pile.isEmpty()) {
            Card card = pile.peek();
            Stack<Card> suit = newsuit.get(command.getIndex());
            if (suit.isEmpty() || (!suit.isEmpty() && suit.peek().ifNextInSet(card))) {
                isPossible = true;
            }
        }
    } else if (command.Moveingtolane() && command.MoveingToSuit()) {
        Stack<Card> lane = newlanes.get(command.getingFromIndex());
        if (!lane.isEmpty()) {
            Card card = lane.peek();
            Stack<Card> suit = newsuit.get(command.getIndex());
            if (suit.isEmpty() || (!suit.isEmpty() && suit.peek().ifNextInSet(card))) {
                isPossible = true;
            }
        }
    } else if (command.Moveingfromsuit() && command.MoveinginsideLane()) {
        Stack<Card> suit = newsuit.get(command.getingFromIndex());
        if (!suit.isEmpty()) {
            Card card = suit.peek();
            Stack<Card> lane = newlanes.get(command.getIndex());
            if (lane.isEmpty() || (!lane.isEmpty() && lane.peek().ifNextInLane(card))) {
                isPossible = true;
            }
        }
    } else {
        List<Card> fromLane = newlanes.get(command.getingFromIndex());
        if (fromLane.size() >= command.Numberofcardstomove() && fromLane.get(fromLane.size() -
command.Numberofcardstomove()).ifFaceUp()) {
            Card card = fromLane.get(fromLane.size() - command.Numberofcardstomove());
            Stack<Card> toLane = newlanes.get(command.getIndex());
            if (toLane.isEmpty() || (!toLane.isEmpty() && toLane.peek().ifNextInLane(card))) {
                isPossible = true;
            }
        }
    }
    return isPossible;
}

    public void move(Command command) {
        // Execute the given move
        reveledcards = false;
        if (command.Moveingtopile() && command.MoveinginsideLane()) {
            Card card = pile.pop();
            newlanes.get(command.getIndex()).push(card);
        } else if (command.Moveingtopile() && command.MoveingToSuit()) {
            Card card = pile.pop();
            newsuit.get(command.getIndex()).push(card);
        } else if (command.Moveingtolane() && command.MoveingToSuit()) {
            Stack<Card> lane = newlanes.get(command.getingFromIndex());
            Card card = lane.pop();
            if (!lane.isEmpty() && lane.peek().ifFaceDown()) {
                lane.peek().CheckFaceUp();
                reveledcards= true;
            }
            newsuit.get(command.getIndex()).push(card);
```

```
        } else if (command.Moveingfromsuit() && command.MoveinginsideLane()) {
            Card card = newsuit.get(command.getingFromIndex()).pop();
            newlanes.get(command.getIndex()).push(card);
        } else {
            List<Card> fromLane = newlanes.get(command.getingFromIndex());
            int firstCardToMoveIndex = fromLane.size() - command.Numberofcardstomove();
            for (int i = 0; i < command.Numberofcardstomove(); i++) {
                Card card = fromLane.get(firstCardToMoveIndex);
                fromLane.remove(firstCardToMoveIndex);
                newlanes.get(command.getIndex()).push(card);
            }
            if (!fromLane.isEmpty() && fromLane.get(fromLane.size() - 1).ifFaceDown()) {
                fromLane.get(fromLane.size() - 1).CheckFaceUp();
                reveledcards = true;
            }
        }
    }
    public boolean GameOver() {
        // Check if the game is over
        for (Stack<Card> suit : newsuit) {
            if (suit.size() != Suitsize) {
                return false;
            }
        }
        return true;
    }

    public int maxLaneSize() {
        // Get the maximum size of the lanes
        int maxLaneSize = 0;
        for (Stack<Card> lane : newlanes) {
            if (lane.size() > maxLaneSize) {
                maxLaneSize = lane.size();
            }
        }
        return maxLaneSize;
    }
    // Getters for various components of the game
    public Deck getDeck() {

        return newdeck;
    }

    public Stack<Card> getPile() {
        return pile;
    }

    public Stack<Card> getLane(int index) {
        return newlanes.get(index);
    }

    public Stack<Card> getSuit(int index) {
        return newsuit.get(index);
    }

    public boolean wasCardRevealed() {
        // Check if cards were revealed in the last move
        return reveledcards;
    }
    public static void main(String[] args) {
```

```
        // Main method to run the game
    PatienceGame game = new PatienceGame();
    Score score = new Score();
    View view = new View();
    view.displayWelcome();
    Command command;
    do {
      view.displayScore(score);
      view.displayBoard(game);
      boolean commandDone = false;
      do {
        command = view.getUserInput();
        if (command.Drawing()) {
          if (game.Possiblelityofmove()) {
            game.takingdraw();
            score.drawsPlayed(command);
            commandDone = true;
          } else {
            view.displayCommandNotPossible();
          }
        } else if (command.Moveing()) {
          if (game.moveIsPossible(command)) {
            game.move(command);
            score.movePlayed(command, game);
            commandDone = true;
          } else {
            view.displayCommandNotPossible();
          }
        } else if (command.Quitting ()) {
          commandDone = true;
        }
      } while (!commandDone);
    } while (!command.Quitting() && !game.GameOver());
    if (game.GameOver()) {
      view.displayScore(score);
      view.displayBoard(game);
      view.displayGameOver();
    } else {
      view.displayQuit();
    }
  }
}
```

In this Patience Game following things are there

Patience Game:-

The class has a constructor PatienceGame(), which is used to initialize a new game. It creates a shuffled deck, distributes cards to the lanes (with one face-up card at the bottom of each lane), and initializes the suit stacks and other variables.

```
enum Suitcolour{
class Card {
            // Class to represent a playing card
            private Suit set;
            private Rank variable;
            private boolean Upperface;

            public Card() {
                        set =Suit.HEART;
                        variable =Rank.QUEEN;
                        Upperface=true;
            }
            public Card(Suit set,Rank variable,boolean Uface) {
                         this.set = set;
                    this.variable = variable;
                    Upperface = Uface;
            }
            public void CheckFaceUp() {
                        // Turn the card face up
        Upperface = true;
    }

    public void CheckFaceDown() {
            // Turn the card face down
        Upperface = false;
    }

    public boolean ifFaceUp() {
            // Check if the card is face up
        return Upperface;
    }

    public boolean ifFaceDown() {
            // Check if the card is face down
        return !Upperface;
    }
    public Suit getset() {
            // Get the suit of the card
        return set;
    }

    public Rank getVariable() {
            // Get the rank of the card
        return variable;
    }


    private boolean ifSameset(Card card) {
             // Check if two cards have the same suit
        return set == card.getset();
    }

    private boolean ifNextLowerVariable(Card card) {
             // Check if the card is the next lower rank in a lane
        return this.variable.ordinal() - 1 == card.getVariable().ordinal();
    }

    private boolean ifNextHigherVariable(Card card) {
             // Check if the card is the next higher rank in a lane
        return this.variable.ordinal() + 1 == card.getVariable().ordinal();
```

```
    }

    private boolean ifDifferentColour(Card card) {
            // Check if two cards have different colors
        return set.getColour() != card.getset().getColour();
    }

    public boolean ifNextInLane(Card card) {
            // Check if the card can be placed on top of another card in a lane
        return ifDifferentColour(card) && ifNextLowerVariable(card);
    }

    public boolean ifNextInSet(Card card) {
            // Check if the card can be placed on top of another card in a suit stack
        return ifSameset(card) && ifNextHigherVariable(card);
    }

    public String toString() {
        // Convert the card to a string for display
        if (Upperface) {
            return set.toString() + variable.toString();
        } else {
            return "*x*";
        }
    }

}
```

Class Card:-

Overall, this class and the enumerations are used to represent and manage individual playing cards within a card game, and they provide methods to check card properties, such as rank, suit, and orientation (face-up or face-down). These elements are fundamental for implementing card-based games like the Patience game mentioned in the previous response.

```
class Deck extends Stack<Card> {
            // Class to represent a deck of cards

        public Deck() {
            initializeDeck();
        }

        public void shuffleDeck() {
            // Shuffle the deck
            Collections.shuffle(this);
        }

        private void initializeDeck() {
            // Initialize the deck with all cards
            for (Suit suit : Suit.values()) {
                for (Rank rank : Rank.values()) {
                    Card card = new Card(suit, rank, false);
                    this.add(card);
                }
            }
        }
    }
```

Class Deck -

The Deck class is an essential component in card games because it represents the initial state of the game, where all the cards are present but not necessarily in a specific order. Game logic can use this class to create, shuffle, and manage a deck of cards, which is then used to deal cards to players, draw cards during the game, and more.

```
class Command {
        // Class to represent a user command
        private enum Commandtype{DRAW,MOVE,QUIT}

        private Commandtype commandtype ;
        private char moveFrom,moveTo;
        private String allcards;

        public Command(String User_inp) {

                String Formatinp = User_inp.trim().toUpperCase();
                if (Formatinp.equals("Q"))
                {
                        commandtype=Commandtype.QUIT;
                }
                else if (Formatinp.equals("M"))
                {
                        commandtype = Commandtype.MOVE;
              moveFrom = Formatinp.charAt(0);
              moveTo = Formatinp.charAt(1);
              if (Formatinp.length() == 2) {
                 allcards = "";
              }
              else {
                allcards = Formatinp.substring(2);
              }
                }
                else {
                        commandtype=Commandtype.DRAW;
                }
        }
        public static boolean CheckingValid(String inp) {
                // Check if the user input is a valid command
      String cleanip = inp.trim().toUpperCase();
      return cleanip.equals("Q") ||
           cleanip.equals("D") ||
           cleanip.matches("[P1-7DHCS][1-7DHCS][0-9]*");
    }
         public boolean Quitting() {
                // Check if the command is to quit the game
            return commandtype == Commandtype.QUIT;
          }

          public boolean Drawing() {
                 // Check if the command is to draw a card
            return commandtype == Commandtype.DRAW;
          }

          public boolean Moveing() {
                // Check if the command is to move a card
            return commandtype == Commandtype.MOVE;
          }
```

```
public boolean Moveingtopile() {
        // Check if the move is from pile to a lane
    return moveFrom == 'P';
}

public boolean Moveingtolane() {
    return Character.toString(moveFrom).matches("[1-7]");
}

public boolean Moveingfromsuit() {
        // Check if the move is from pile to a suit stack
    return Character.toString(moveFrom).matches("[DHCS]");
}

private int indexingrthesuit (char Character) {

        return switch (Character) {
        case 'D'->0;
        case 'H'->1;
        case 'C'->2;
        case 'S'->3;
        default ->0;
        };

}
public int getingFromIndex() {
        // Get the index for the source of the move
    if (Moveingtolane()) {
        return Character.getNumericValue(moveFrom) - 1;
    } else {
        return indexingrthesuit(moveFrom);
    }
}

public boolean MoveinginsideLane() {
        // Check if the move is from a lane to a lane
    return Character.toString(moveTo).matches("[1-7]");
}
// Get the index for the destination of the move
public int getIndex() {
        // Get the index for the destination of the move
    if (MoveinginsideLane()) {
        return Character.getNumericValue(moveTo) - 1;
    } else {
        return indexingrthesuit(moveTo);
    }
}

public boolean MoveingToSuit() {
        // Check if the move is from a suit stack to a lane
    return Character.toString(moveTo).matches("[DHCS]");
}

public int Numberofcardstomove() {
        // Get the number of cards to move
    if (allcards.equals(" ") || allcards.equals("1")) {
        return 1;
    } else {
        return Integer.valueOf(allcards);
    }
```

```
            }
    }
```

Class Command:-

The Command class serves as a mechanism to interpret and encapsulate user commands for a card game. The Command class essentially serves as an interface for interpreting and understanding user commands, making it easier to implement game logic by processing user instructions effectively.

```
class Score{
            // Class to keep track of the game score
            private int point =0;
            private int turns=0;

            public int getingpoints() {
                        // Get the current score
          return point;
       }

       public int getingTurns() {
            // Get the number of turns played

          return turns;
       }
       public void drawsPlayed(Command command) {
            // Update the score when draws are played
          turns++;
       }
       public void movePlayed(Command command, PatienceGame game) {
            // Update the score when a move is played
          turns++;
          if (command.Moveingtopile() && command.Moveingtolane()) {
             point += 5;
          } else if (command.Moveingtopile() && command.MoveingToSuit()) {
             point += 10;
          } else if (command.Moveingtolane() && command.MoveinginsideLane() &&
game.wasCardRevealed()) {//last function of patiencwegame
             point += 5;
          } else if (command.Moveingtolane() && command.MoveinginsideLane()) {
             point +=-10;
          } else {
             point += 20;
          }
       }
    }
```

Class Score :-

The Score class is responsible for maintaining and updating the game score based on the player's actions during the card game. the Score class provides a mechanism for tracking and updating the player's score based on their actions in the card game. It keeps count of the number of turns played and adjusts the score accordingly. The score updates are determined by the specific moves made during the game and the game's rules.

```
class Board{
            // Class to represent the game board
        public static final int LANESNumber = 7;
        public static final int SUITSNumber = Suit.values().length;
        public static final int SUITSize = Rank.values().length;

        private Deck Allcards;
        private Stack<Card> Drawnpile;
        private List<List<Card>> coloum ;
        private List<Stack<Card>> suits;
        private boolean Checkreveledcards;

        public Board() {
            Allcards=new Deck();
            Allcards.shuffleDeck();
            Drawnpile=new Stack<>();
            coloum=new ArrayList<>(LANESNumber);
              for (int i = 0; i <LANESNumber ; i++) {
                coloum.add(new ArrayList<>());
                for (int j = 0; j < i; j++) {
                    coloum.get(i).add(Allcards.pop());
                }
                Card card = Allcards.pop();
                card.CheckFaceUp();
                coloum.get(i).add(card);
            }
              suits =  new ArrayList<>(SUITSNumber);
            for (int i = 0; i < SUITSNumber; i++) {
                suits.add(new Stack<>());
            }
            Checkreveledcards= false;
          }
        public boolean Drawpossibailitycheck() {
            // Check if drawing a card is possible
           return !Allcards.isEmpty() || !Drawnpile.isEmpty();
        }
        public void draw() {
            // Draw a card from the deck
           if (!Allcards.isEmpty()) {
              Card card = Allcards.pop();
              card.CheckFaceUp();
              Drawnpile.add(card);
           } else {
              int numberOfCards = Drawnpile.size();
              for (int i = 0; i < numberOfCards; i++) {
                 Card card = Drawnpile.pop();
                 card.CheckFaceDown();
                 Allcards.add(card);
              }
           }
        }
        public boolean movePossibity(Command command) {
            // Check if a move is possible
           boolean isPossible = false;
           if (command.Moveingtopile() && command.Moveingtolane()) {
              if (!Drawnpile.isEmpty()) {
                 Card card = Drawnpile.peek();
                 List<Card> lane = coloum.get(command.getIndex());
```

```
            if (lane.isEmpty() || (!lane.isEmpty() && lane.get(lane.size() - 1).ifNextInLane(card))) {
                isPossible = true;
            }
        }
    } else if (command.Moveingtopile() && command.MoveingToSuit()) {
        if (!Drawnpile.isEmpty()) {
            Card card = Drawnpile.peek();
            Stack<Card> suit = suits.get(command.getIndex());
            if (suit.isEmpty() || (!suit.isEmpty() && suit.peek().ifNextInSet(card))) {
                isPossible = true;
            }
        }
    } else if (command.Moveingfromsuit() && command.MoveingToSuit()) {
        List<Card> lane = coloum.get(command.getingFromIndex());
        if (!lane.isEmpty()) {
            Card card = lane.get(lane.size() - 1);
            Stack<Card> suit = suits.get(command.getIndex());
            if (suit.isEmpty() || (!suit.isEmpty() && suit.peek().ifNextInSet(card))) {
                isPossible = true;
            }
        }
    } else if (command.Moveingfromsuit() && command.MoveinginsideLane()) {
        Stack<Card> suit = suits.get(command.getingFromIndex());
        if (!suit.isEmpty()) {
            Card card = suit.peek();
            List<Card> lane = coloum.get(command.getIndex());
            if (lane.isEmpty() || (!lane.isEmpty() && lane.get(lane.size() - 1).ifNextInLane(card))) {
                isPossible = true;
            }
        }
    } else {
        // lane to lane
        List<Card> fromLane = coloum.get(command.getingFromIndex());
        if (fromLane.size() >= command.Numberofcardstomove() &&
                fromLane.get(fromLane.size() - command.Numberofcardstomove()).ifFaceUp()) {
            Card card = fromLane.get(fromLane.size() - command.Numberofcardstomove());
            List<Card> toLane = coloum.get(command.getIndex());
            if (toLane.isEmpty() || (!toLane.isEmpty() && toLane.get(toLane.size() - 1).ifNextInLane(card))) {
                isPossible = true;
            }
        }
    }
    return isPossible;
}
public boolean GameOver() {
    // Check if the game is over
    for (Stack<Card> suit : suits) {
        if (suit.size() != SUITSize) {
            return false;
        }
    }
    return true;
}

public int maxLaneSize() {
        // Get the maximum size of the lanes
    int maxLaneSize = 0;
    for (List<Card> lane : coloum) {
        if (lane.size() > maxLaneSize) {
            maxLaneSize = lane.size();
```

```
            }
        }
        return maxLaneSize;
    }
    // Getters for various components of the game
    public Deck getDeck() {

        return Allcards;
    }

    public Stack<Card> getPile() {
        return Drawnpile;
    }

    public List<Card> getLane(int index) {
        return coloum.get(index);
    }

    public Stack<Card> getSuit(int index) {
        return suits.get(index);
    }

    public boolean wasCardReveal() {
            // Check if cards were revealed in the last move
        return Checkreveledcards;
    }

    }
```

Class Board :-

The Board class is responsible for representing and managing the game board, which includes the deck of cards, drawn pile, columns (lanes), and suit stacks. The game board keeps track of card positions and game state.

```
lass View {

    public void displayWelcome() {
            // Display a welcome message
        System.out.println("**********Patience Game**********");
        System.out.println("Instruction :-");
        System.out.println("Press D - Drawing  a card from the deck");
        System.out.println("Press P1-7 - Moveing  card(s) from pile to a lane");
        System.out.println("Press D1-7 - Moveing  card(s) from pile to a suit");
        System.out.println("Press 1-7D - Moving card(s) from a lane to a lane");
        System.out.println("Press D1-7HCS - Moveing card(s) from a suit to a lane");
        System.out.println("Press Q - Quiting the game");
    }
    public void displayScore(Score score) {
          // Display the current score
        System.out.println("Score: " + score.getingpoints());
        System.out.println("Number of turns: " + score.getingTurns());
    }
    public void displayBoard(PatienceGame game) {
            // Display the game board
        System.out.println("Deck: " + (game.getDeck().isEmpty() ? "Empty" : "Not Empty"));
```

```
        System.out.println("Pile: " + (game.getPile().isEmpty() ? "Empty" : "Not Empty"));
        for (int i = 0; i < Board.SUITSNumber; i++) {
            System.out.print("Suit " + (i + 1) + ": ");
            Stack<Card> suit = game.getSuit(i);
            if (suit.isEmpty()) {
                System.out.println("Empty");
            } else {
                System.out.println(suit.peek().toString());
            }
        }
        for (int i = 0; i < Board.LANESNumber; i++) {
            System.out.print("Lane " + (i + 1) + ": ");
            List<Card> lane = game.getLane(i);
            for (Card card : lane) {
                System.out.print(card.toString() + " ");
            }
            System.out.println();
        }
    }
    public void displayCommandNotPossible() {
         // Display the message for an impossible command
        System.out.println("Instruction is not possible. Please try again.");
    }

    public void displayGameOver() {
         // Display the game over message
        System.out.println("**********Congratulations! You are a Winner **********");
    }

    public void displayQuit() {
          // Display the quit message
        System.out.println("******You exited the game. Good luck next time!*****");
    }

    public Command getUserInput() {
         // Get user input
        Scanner scanner = new Scanner(System.in);
        String input;
        do {
            System.out.print("Enter a command: ");
            input = scanner.nextLine().trim();
            if (!Command.CheckingValid(input)) {
                System.out.println("Invalid command. Please try again.");
            }
        } while (!Command.CheckingValid(input));
        return new Command(input);
    }
}
```

Class View:-

The View class is responsible for creating a user-friendly interface for the Patience Game, providing instructions, displaying the game state, and collecting user commands to interact with the game. It interacts with the game logic implemented in the PatienceGame class and the scoring system in the Score class to facilitate gameplay.

Checklist:-

Class Name	Functionality	Error explanation
Patience Game	Functional	N/A
View	Functional	N/A
Board	Functional	N/A
Score	Partially Functional	Sometimes the score is getting updated properly and when the Inconsistent error is coming the it is not getting updated .
Deck	Functional	N/A
Card	Functional	N/A
Command	Partially Functional	This class is not shifting the cards properly between the lanes . Alt the methods are not getting called properly .

Inconsistent Issue:-The code is running fine in VS Code sometimes and sometimes cards are not moving between the lanes .I am not able to find the exact issue with the code but I am hoping that there is some issue with the command class. In the dew course I will try to fix this issue .

All classes are getting compiled but there is some logic error which I will fix.