
# INTERNATIONAL UNIVERSITY
# VIETNAM NATIONAL UNIVERSITY
# SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

---

## Object–Oriented Programming Final Report

Semester 1, 2023- 2024

Instructor: Tran Thanh Tung

# BlackJack(Xi Dach)

**Link Github:** Drissdo185/BlackJackV2 (github.com)

# 1/Member List:

| Name | Student ID | Contribution |
|---|---|---|
| Do Thanh Dat | ITDSIU21079 | 30% |
| Nguyen Ba Duy | ITDSIU21014 | 30% |
| Ngo Hoang Thanh | ITDSIU21119 | 20% |
| Pham Huynh Thanh Quan | ITDSIU21110 | 20% |

# 2/BlackJack Data(Code, Relate Information):

- **Introduction**

This project was designed based on Object – Oriented Programming method in Java language. So this has solved some problems that occurred while building by a common Procedural – Oriented Method:

- The code is clear, easy to understand, and concise.
- The project is a unified logic system with many related - Classes combined together.
- Each Class has many Methods which take on different behaviors of their own class. The ability to reuse the resources.

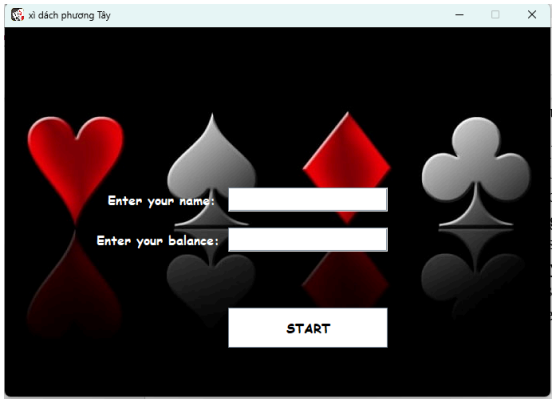
The aim of this undertaking is to create a simple game rooted in object-oriented principles. Our team opted to adapt the game "Blackjack" using this methodology. The game will distinctly exhibit object-oriented characteristics, showcasing the

interplay of classes and objects. In this game, players aim to reach a card total close to or exactly 21 to emerge victorious. It's important to note that our game is a work in progress, and not all functions are currently available for demonstration.

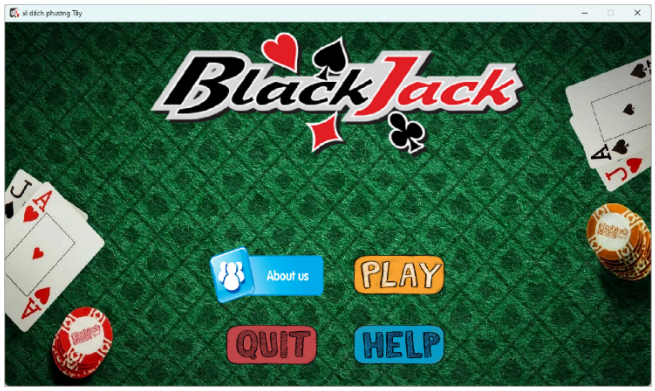
In addition to the previously outlined requirements, our intention in undertaking this project is to apply and reinforce the various techniques we've learned throughout the course. This includes hands-on experience with working with Classes and objects, and logically implementing problem-solving through Object-Oriented Programming (OOP) methods. Despite acknowledging the potential risks and challenges associated with venturing beyond the confines of our university syllabus, we are motivated to challenge ourselves. This endeavor serves as an opportunity to familiarize ourselves with the real-world pressures we may encounter in future workplaces. In essence, our ultimate goal is to progress further along our path towards becoming proficient and advanced software developers.

- **Game BlackJack:**

Login




Display




Play Game:


- **Introduction to gameplay**

**1. GOAL:**

Aim to have a hand value close to 21 without going over.

**2. CARD VALUES:**

- Number cards are face value.
- Face cards (J, Q, K) are 10 points.
- Aces can be 1 or 11 points.

**3. THE DEAL:**

- Players get two cards.
- Dealer has face-down cards.

**4. TURN:**

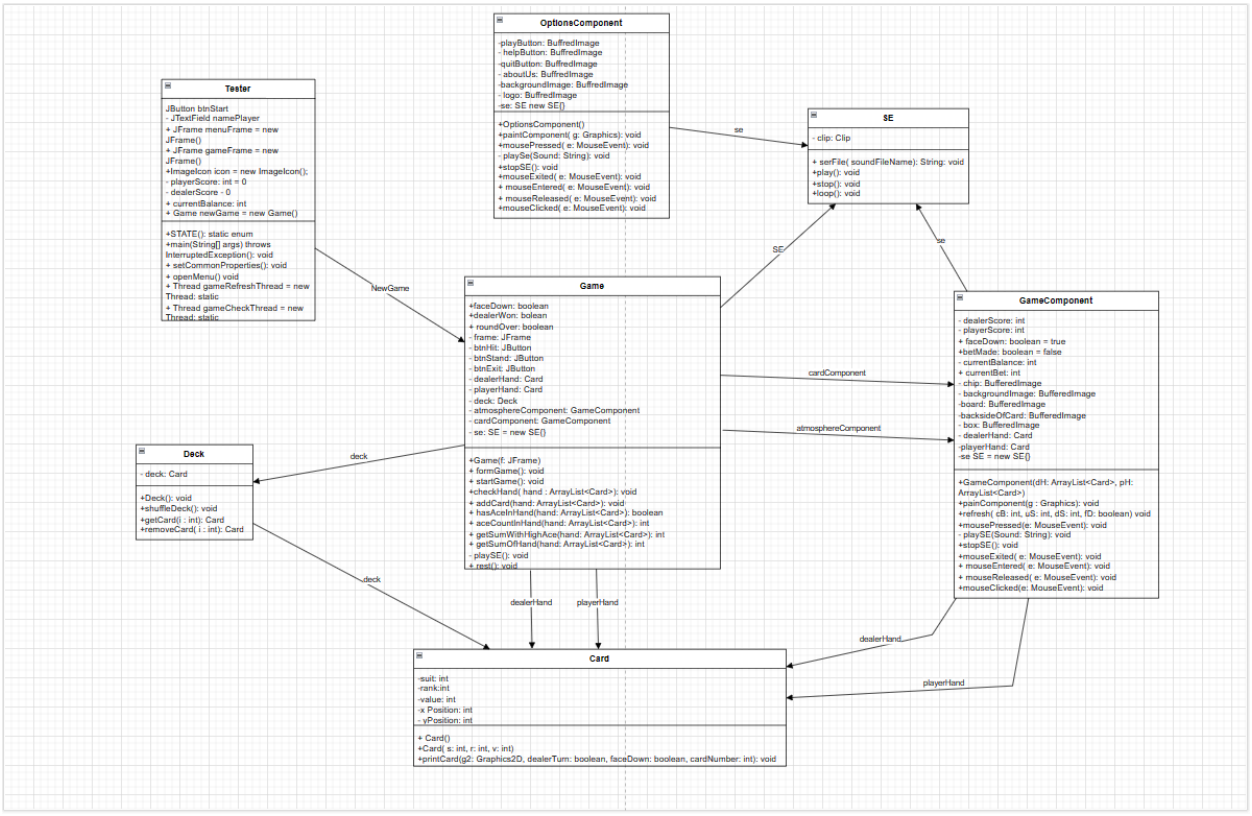
- "Hit" for more cards or "stand" to keep the current hand.
- Continue until the player stands or busts (exceeds 21).

**5. WINNING & LOSING:**

- Win if closer to 21 than the dealer without busting.
- Bust if the hand exceeds 21.
- Dealer bust means the player wins.
- A tie returns the bet.

# 3/UML Diagram of Blackjack

Access:




## 1. Deck Class

The `Deck` class represents a standard deck of playing cards and provides functionalities for creating, shuffling, and accessing cards in the deck. This

class is designed to be used in card games, such as Blackjack, where a shuffled and managed deck of cards is essential.

**Attributes:**

1. deck: An `ArrayList` of `Card` objects, representing the collection of cards in the deck.

Constructors:

1. Default Constructor:

- Initializes the `deck` attribute by populating it with a standard deck of 52 playing cards.

- Cards are created with suits ranging from 0 to 3 and ranks from 0 to 12.

- Special consideration is given for Ace (rank 0), assigning a value of 11.

**Methods:**

1. shuffleDeck():

- Utilizes the `Collections.shuffle` method to randomize the order of cards in the deck.

- Provides a mechanism for shuffling the deck, ensuring randomness in card distribution.

2. getCard(int i):

- Retrieves a specific card from the deck based on its index.

- Enables access to individual cards in the deck by providing the index.

3. removeCard(int i):

- Removes and returns a specific card from the deck based on its index.

- Supports the removal of cards, which is useful for dealing cards in card games.

**Considerations:**

1. Card Representation:

   - Assumes the existence of a `Card` class with appropriate attributes (suit, rank, value) and methods.

2. Ace Handling:

   - Assigns a value of 11 to Ace cards, a common convention in Blackjack.

   - This may be adjusted based on specific game rules.

3. Card Shuffling:

   - Utilizes the `Collections.shuffle` method for shuffling, providing a standard and efficient way to randomize card order.

4. Deck Size:

   - Creates a standard deck with 52 cards, as per traditional playing card standards.

## 2. Card Class

The `Card` class represents a playing card in a standard deck, encapsulating properties such as suit, rank, value, and position. It provides methods for retrieving these properties and includes a method for rendering the card on a graphics context.

**Attributes:**

1. suit: An integer representing the suit of the card.

2. rank: An integer representing the rank of the card.

3. value: An integer representing the numerical value of the card.

4. xPosition, yPosition: Integers indicating the x and y coordinates for rendering the card on a graphics context.

**Constructors:**

1. Default Constructor:

- Initializes suit, rank, and value to 0.

2. Parameterized Constructor:

- Accepts parameters for suit, rank, and value to initialize the card with specific values.

**Methods:**

1. getSuit():

- Returns the suit of the card.

2. getRank():

- Returns the rank of the card.

3. getValue():

- Returns the numerical value of the card.

4. printCard(Graphics2D g2, boolean dealerTurn, boolean faceDown, int cardNumber):

- Accepts a `Graphics2D` object for rendering, along with boolean flags indicating if it's the dealer's turn and if the card should be face down.

- Reads card images from a sprite sheet and back of a card image from files.

- Positions and renders the card based on the provided parameters.

- Supports rendering face-down cards and different positions for dealer and player.

**File I/O:**

1. Image Loading:

- Utilizes the `ImageIO` class to read card images from a sprite sheet and the back of a card image from files.

- Assumes the existence of image files like "cardSpriteSheet.png" and "backsideOfACard.jpg."

Graphics Rendering:

1. Rendering Cards:

- Utilizes the `Graphics2D` object to draw the card on the specified coordinates.

- Handles face-down cards by rendering the back of the card image.

- Supports dynamic positioning based on whether it's the dealer's turn and the card's position in the hand.

**Dependencies:**

1. Image Files:

- Depends on image files (e.g., "cardSpriteSheet.png" and "backsideOfACard.jpg") for rendering card faces and backs.

**Considerations:**

1. Assumptions:

- Assumes the existence and correctness of image files.

- Assumes a standard deck of cards with 4 suits and 13 ranks.

## 3. Game Class

The `Game` class is the central component managing the game logic and user interface for a card game, presumably Blackjack. It coordinates interactions between the deck, players, and graphical components. The class encapsulates features such as player hands, buttons, round outcomes, and user interface components.

**Attributes:**

1. dealerHand, playerHand: ArrayLists representing the hands of the dealer and player, respectively.

2. faceDown: A boolean indicating whether the dealer's first card should be face-down.

3. dealerWon, tie: Boolean flags indicating the outcome of the round (dealer won, tie).

4. roundOver: A volatile boolean tracking whether the current round has concluded.

5. frame: A JFrame for displaying the game interface.

6. deck: An instance of the `Deck` class for managing the deck of cards.

7. atmosphereComponent, cardComponent: Instances of `GameComponent` for rendering graphical elements.

8. btnHit, btnStand, btnExit: JButtons for player actions (hit, stand, exit).

9. playerName: A String storing the player's name.

10. se: An instance of the `SE` class for handling sound effects.

**Constructors:**

1. Parameterized Constructor:

- Initializes the `deck`, `dealerHand`, and `playerHand`.

- Sets default values for boolean flags (`faceDown`, `dealerWon`, `tie`, `roundOver`).

- Takes a `JFrame` and player name as parameters.

**Methods:**

1. formGame():

- Configures the JFrame and adds buttons for hit, stand, and exit.

- Initializes `atmosphereComponent` for rendering game components.

- Makes the JFrame visible.

2. startGame():

- Distributes initial cards to the dealer and player.

- Adds action listeners to buttons for hit, stand, and exit.

- Checks for Blackjack conditions and starts the game.

3. checkNumberOfCards(ArrayList<Card> hand):

- Returns the number of cards in a given hand.

4. addCard(ArrayList<Card> hand):

- Adds a card to the specified hand from the deck.

- Updates the `faceDown` flag.

5. checkBJ(ArrayList<Card> hand):

- Checks for Blackjack in a given hand (dealer or player).
- Notifies the outcome if a Blackjack is detected.

6. checkFiveCardD(ArrayList<Card> hand):

- Checks if the dealer has a Five Card Trick.

7. checkFiveCardP(ArrayList<Card> hand):

- Checks if the player has a Five Card Trick.

8. checkFiveCardLP(ArrayList<Card> hand):

- Checks if the player has a Five Card Trick after losing.

9. hasAceInHand(ArrayList<Card> hand):

- Checks if a hand contains an Ace.

10. aceCountInHand(ArrayList<Card> hand):

- Returns the count of Aces in a hand.

11. getSumWithHighAce(ArrayList<Card> hand):

- Calculates the sum of hand values considering Aces as 11.

12. getSumOfHand(ArrayList<Card> hand):

- Calculates the total value of a hand, considering Ace as 11 or 1 based on the total sum.

13. playSE(String Sound):

- Plays a sound effect based on the provided sound file path.

14. stopSE():

- Stops the currently playing sound effect.

15. rest():

- Introduces a pause in the execution (used for smooth transitions).

**User Interface:**

- Configures buttons with appropriate fonts and sizes.

- Provides action listeners for user interactions (e.g., hitting, standing, exiting).

**Sound Effects:**

- Utilizes the `SE` class for handling sound effects.

- Plays sound effects on specific game events (e.g., card draw, round completion).

**Considerations:**

1. Game Logic:

- Manages game flow, including card distribution, player actions, and round outcomes.

- Checks for special conditions such as Blackjack and Five Card Tricks.

2. User Interface:

- Implements a graphical user interface using Swing components.

- Allows player actions through buttons for hit, stand, and exit.

3. Sound Effects:

- Enhances the gaming experience with sound effects for various events.

## ● GameComponent class

The `GameComponent` class extends `JComponent` and implements the `MouseListener` interface in Java Swing. It serves as a graphical component responsible for rendering and interacting with the user interface of a card game, likely Blackjack. This class manages the display of cards, chips, scores, and other UI elements, while also handling mouse events for user interaction.

**Attributes:**

1. dealerHand, playerHand: ArrayLists representing the hands of the dealer and player, respectively.

2. dealerScore, playerScore: Integers representing the scores of the dealer and player.

3. faceDown: A boolean indicating whether the dealer's first card should be face-down.

4. betMade: A static boolean tracking whether a bet has been made.

5. currentBalance, currentBet: Integers representing the current balance and bet amount.

6. se: An instance of the `SE` class for playing sound effects.

7. chip, backgroundImage, board, backsideOfACard, box: Static `BufferedImages` for various graphical elements.

Constructor:

- Accepts dealer and player hands as parameters and initializes scores and other components.

- Adds itself as a `MouseListener` to handle mouse events.

**Methods:**

1. paintComponent(Graphics g):

  - Overrides the `paintComponent` method to customize the rendering of graphical elements on the component.

  - Draws background images, cards, scores, and other UI elements.

2. refresh(int cB, int uS, int dS, boolean fD):

  - Updates the component with new values for the current balance, player score, dealer score, and face-down status.

  - Repaints the component to reflect these changes.

3. mousePressed(MouseEvent e):

- Implements the `MouseListener` interface to handle mouse press events.

- Checks if the player clicks on the chip area to initiate the betting process.

- Displays a dialog for the player to choose a betting amount and updates the balance and current bet accordingly.

- If the bet is valid, triggers the start of the game.

4. playSE(String Sound):

- Utilizes the `SE` class to play sound effects based on the provided sound file path.

5. stopSE():

- Utilizes the `SE` class to stop playing sound effects.

6. mouseExited, mouseEntered, mouseReleased, mouseClicked (Placeholder Methods):

- Placeholder methods for the `MouseListener` interface.

**Graphical Elements:**

- Utilizes static `BufferedImages` for chips, background, board, backside of a card, and a box.

- Renders graphical elements to create an immersive and visually appealing game interface.

**Sound Effects:**

- Implements sound effects for mouse clicks and other game events.

- Enhances the gaming experience with audio cues.

**User Interface:**

- Manages graphical elements and user interaction, creating an engaging user interface for the card game.

**Considerations:**

- Assumes the existence of a `SE` class for handling sound effects.
- Requires specific file paths for images and sound effects.

## ● OptionsComponents class

The `OptionsComponent` class extends `JComponent` and implements the `MouseListener` interface in Java Swing. It is designed to handle the graphical elements and user interactions within the options menu of a card game, likely Blackjack. The class manages the display of buttons such as play, help, quit, and about us, providing users with essential functionalities and information.

**Attributes:**

1. playButton, helpButton, quitButton, aboutUs: Static `BufferedImages` representing graphical elements for buttons and informative sections.

2. backgroundImage, logo: Static `BufferedImages` for background and logo images.

3. se: An instance of the `SE` class for playing sound effects.

**Constructor:**

- Initializes the class and adds itself as a `MouseListener` to handle mouse events.

**Methods:**

1. paintComponent(Graphics g):

- Overrides the `paintComponent` method to customize the rendering of graphical elements on the component.

- Draws background images, logo, buttons for play, help, quit, and about us.

2. mousePressed(MouseEvent e):

- Implements the `MouseListener` interface to handle mouse press events.

- Detects which button is clicked based on mouse coordinates and performs corresponding actions.

- Invokes sound effects and transitions to different states within the game.

3. playSE(String Sound):

- Utilizes the `SE` class to play sound effects based on the provided sound file path.

4. stopSE():

- Utilizes the `SE` class to stop playing sound effects.

**Graphical Elements:**

- Utilizes static `BufferedImages` for buttons (play, help, quit, about us), background, and logo.

- Creates an aesthetically pleasing options menu with clear visual cues.

**Sound Effects:**

- Implements sound effects for mouse clicks to enhance the interactive experience.

**User Interface:**

- Manages graphical elements and user interaction, creating an engaging options menu for the card game.

**Functionality:**

1. Play Button:

   - Initiates the game by transitioning to the game state.

   - Starts game-related threads for refresh and checking.

2. Help Button:

   - Displays a dialog box providing rules and guidelines for playing Blackjack.

3. Quit Button:

   - Exits the application when clicked.

4. About Us Button:

   - Provides information about the developers of the Blackjack game.

**Considerations:**

- Assumes specific file paths for images and sound effects.

- Assumes the existence of a `SE` class for handling sound effects.

- Assumes a coordinated transition between different game states.

# 4/ Concepts:

## 1. Abstraction

From my source code, we do not build any Abstraction class.

## 2. Inheritance

```
You, 2 hours ago | 4 authors (driss and others)
public class GameComponent extends JComponent implements MouseListener {

    private ArrayList<Card> dealerHand;
    private ArrayList<Card> playerHand;
```

- This means that GameComponent is a subclass of JComponent. JComponent is an abstract superclass for all Swing components in Java. By extending the JComponent, GameComponent inherits all fields and methods from the JComponent. It can be used as a Swing Component itself.

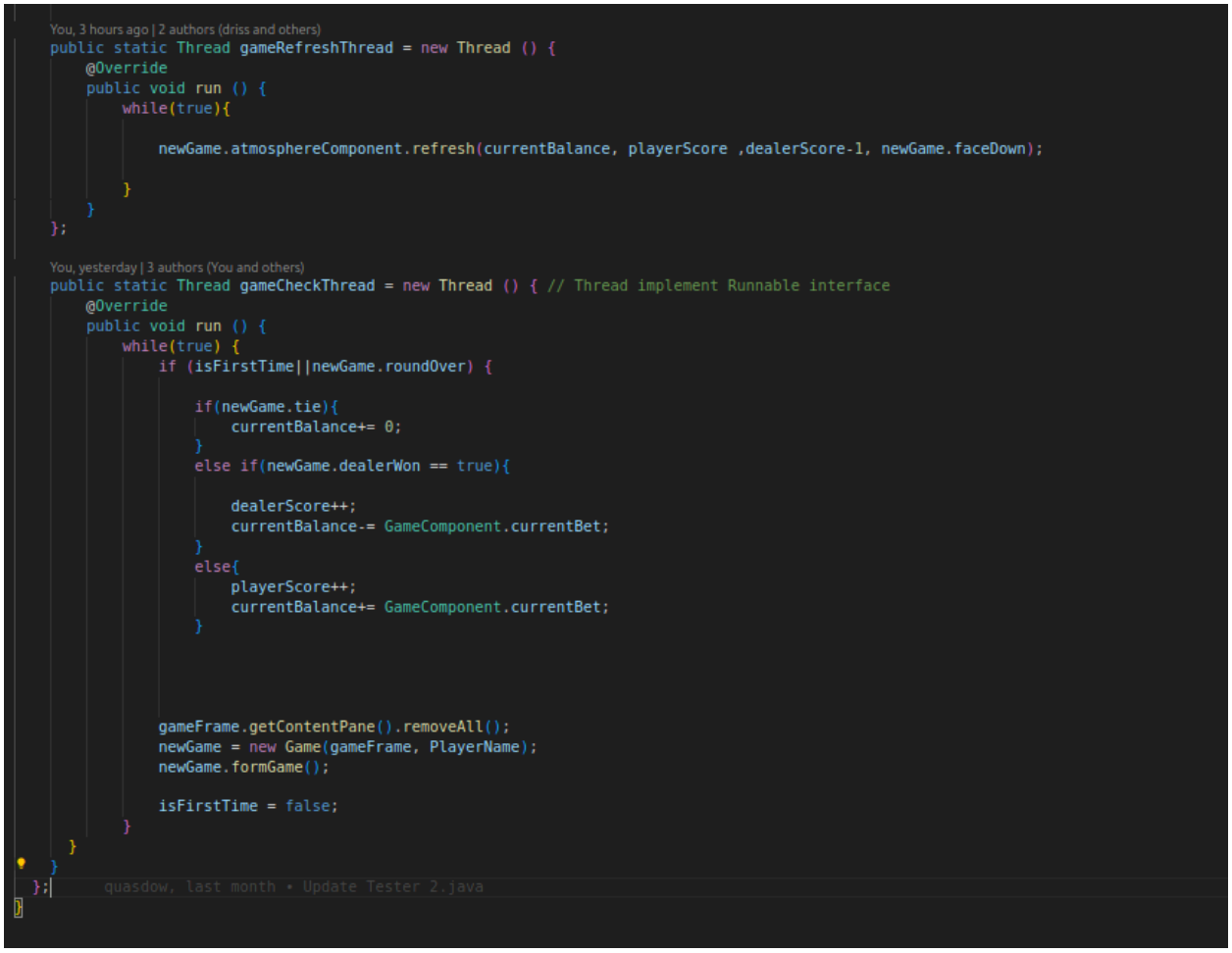
## 3. Encapsulation

```
public class GameComponent extends JComponent implements MouseListener {

    private ArrayList<Card> dealerHand;
    private ArrayList<Card> playerHand;
    private int dealerScore;
    private int playerScore;
    public boolean faceDown = true;
    public static boolean betMade = false;
    private int currentBalance;
    public static int currentBet;
    SE se = new SE();
```

In this example, we set the private (access modifier) for variables, this means that the variable can be used only in this class.

## 4. Polymorphism




Polymorphism is a concept in object-oriented programming that allows objects of different types to be treated as objects of a parent type. It involves overriding (for classes) or implementing (for interfaces) methods in subclasses or classes.

In the provided code, polymorphism is demonstrated through the use of the `Runnable` interface, which is implicitly implemented by the anonymous inner classes that define the `gameRefreshThread` and `gameCheckThread` Threads.

Here's how it works:

- `new Thread() {...}`: This creates an anonymous inner class that extends `Thread`. The `Thread` class itself implements the `Runnable` interface.

- `@Override public void run() {...}`: This is where the `run` method from the `Runnable` interface is being overridden. The `run` method is where you define the code that should be executed in the thread.

So, the polymorphism here is that an instance of the anonymous inner class can be treated as an instance of `Thread` or `Runnable`, and the `run` method can have different behaviors in different `Runnable` objects (i.e., it is polymorphic).

## 5. Design Pattern (Singleton)

```
public static Tester getInstance() {
    if (instance == null) {
        instance = new Tester();
    }
    return instance;      You, 3 days ago • Add design pattern, and change the rule
}


Run | Debug
public static void main(String[] args) throws InterruptedException {
Tester tester = Tester.getInstance();
    if(tester.currentState == STATE.MENU) {
        tester.setCommonProperties();
        tester.openMenu();
    }
}
```

The provided code is in Java and it demonstrates the Singleton design pattern and a simple usage of it.

`public static Tester getInstance() {...}`: This method is part of the Singleton design pattern. It ensures that only one instance of the `Tester` class is created. If an instance already exists, it returns that instance. If not, it creates a new instance.

`public static void main(String[] args) throws InterruptedException {...}`: This is the entry point for the Java application.

`Tester tester = Tester.getInstance();`: This line gets the singleton instance of the Tester class.

`if(tester.currentState == STATE.MENU) {...}`: This conditional statement checks if the current state of the `tester` object is `STATE.MENU`. If it is, it sets some common properties and opens the menu. The `setCommonProperties()` and `openMenu()` methods are presumably defined elsewhere in the `Tester` class.

# 5/ Conclusion:

The Blackjack game is constructed using object-oriented programming, providing a transparent demonstration of OOP principles such as polymorphism, inheritance, encapsulation, and abstraction. The tight and systematic linkage between classes and objects is evident, showcasing the effective application of OOP concepts. Additionally, undertaking this project offers the valuable opportunity to extend our learning beyond the course boundaries, acquiring knowledge that goes beyond the predefined curriculum. This expansion of knowledge is recognized as a crucial aspect of our project implementation.