Blackjack!

Final Report

Developers

Murat Aitov

Jacob Jordan

Gavin Tate

Course: CPSC 224 - Software Development

I. Introduction and Project Description

This document serves to illustrate the details of the project in a more intimate and deep fashion. It will follow from beginning, middle, to end of our project development while including details of our members, struggles faced, lessons learned, and a great foundation to grow and improve in the future. Not only does this document serve as a receipt for the professor, but a true moment of reflection through the process.

Our project was creating a functional and visually appealing game of casino style Blackjack. Blackjack is a gambling game with the goal of hitting a score of 21 without exceeding it. The player plays against the dealer, where both are dealt two cards to begin, and can use a variety of betting options to optimize profit. An Ace card holds a value of 11 if the total score is $\leq$ 21 or holds a value of 1 if the value of 11 forces the hand to exceed 21. Whoever scores closest to 21 will win their bet, and new hands will be dealt until the user wishes to stop playing. Key parts of the project will be individual card values, deck shuffling, dealing, betting, outcomes, and an appealing GUI.

II. Team Members - Bios and Project Roles

- Gavin Tate is a computer science student with previous experience in marketing, visual design, and gambling. He has coded in C++, Python, and Java. His interests include software development and data science. Previous projects include programs for nutritional trends, a game of Crabs, and a simple baseball game. For the project, his responsibilities were planning, organization, creating card classes and methods, and title screen GUI.
- Murat Aitov is a junior majoring in computer science. He has previous experiences coding for a bank in Russia and a variety of languages ranging from C++, Java, Python, and even creating his own language. His previous projects include game projects for previous courses and predicting the next US presidents using trends and historical data. For his responsibilities on the project, he coded much of the logic while implementing new features and classes as needed.
- Jacob Jordan is a sophomore computer science student. He has previous experience in data algorithms, math, and game design. He has programming experience with C++, Python, and Java. His responsibilities included coding the unit testing and user testing.

III. Project Requirements

This section will vaguely outline the essential pieces to the game's functionality. A wide range of classes are required along with methods to assure they work with each other. For a functioning game of Blackjack to operate, classes of cards, players, currency, betting, outcome display, and a visually appealing GUI.

- Initialize cards, card values, deck, multiple decks, shuffling, and reshuffling.
- Cards must be dealt off the top and in the correct order.
- Money + betting: Initial amount, subtracting and adding winnings/losses.
- Betting options: hit, stand, double, split.

- Player(s) + Dealer: Name, balance, hand, winning/losing. The dealer must deal cards and play according with classic casino rules.
- GameState: currentState, changeState(newState)
- Game: List <player>, Deck, Dealer, startGame, checkWinners,
- GUI: Title screen, game screen, and pop-up panels to prompt user responses.

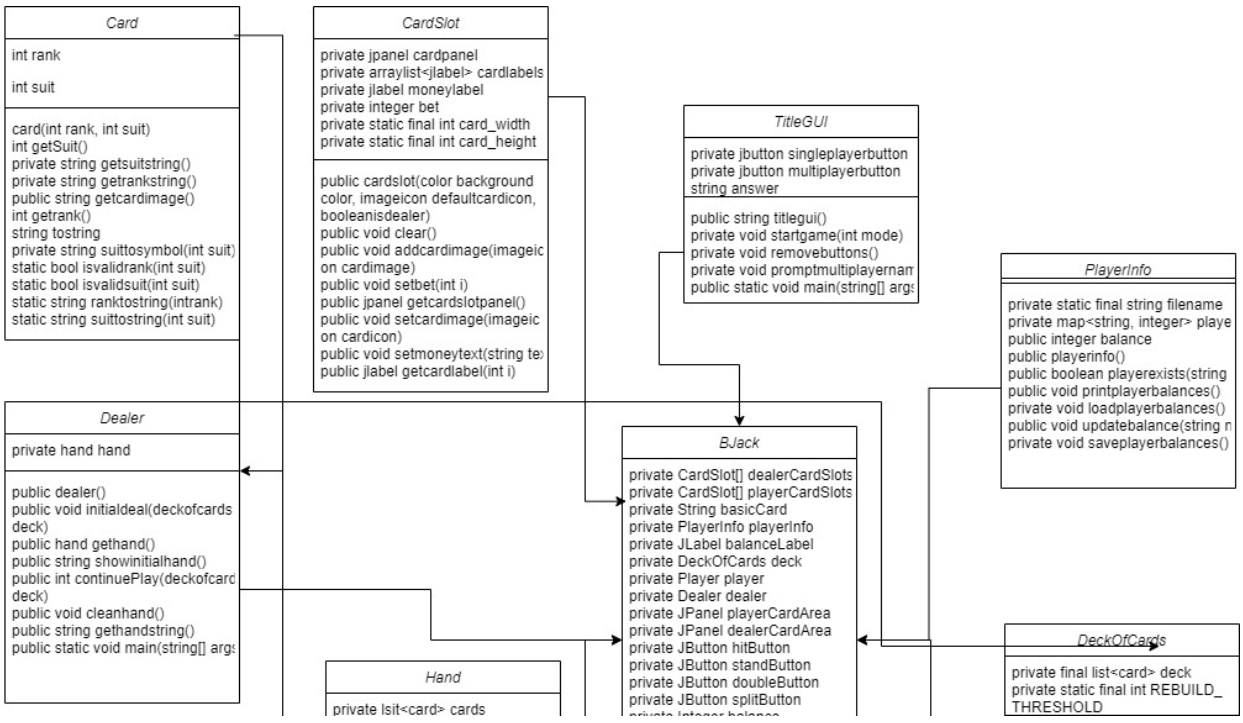
IV. Solution Approach

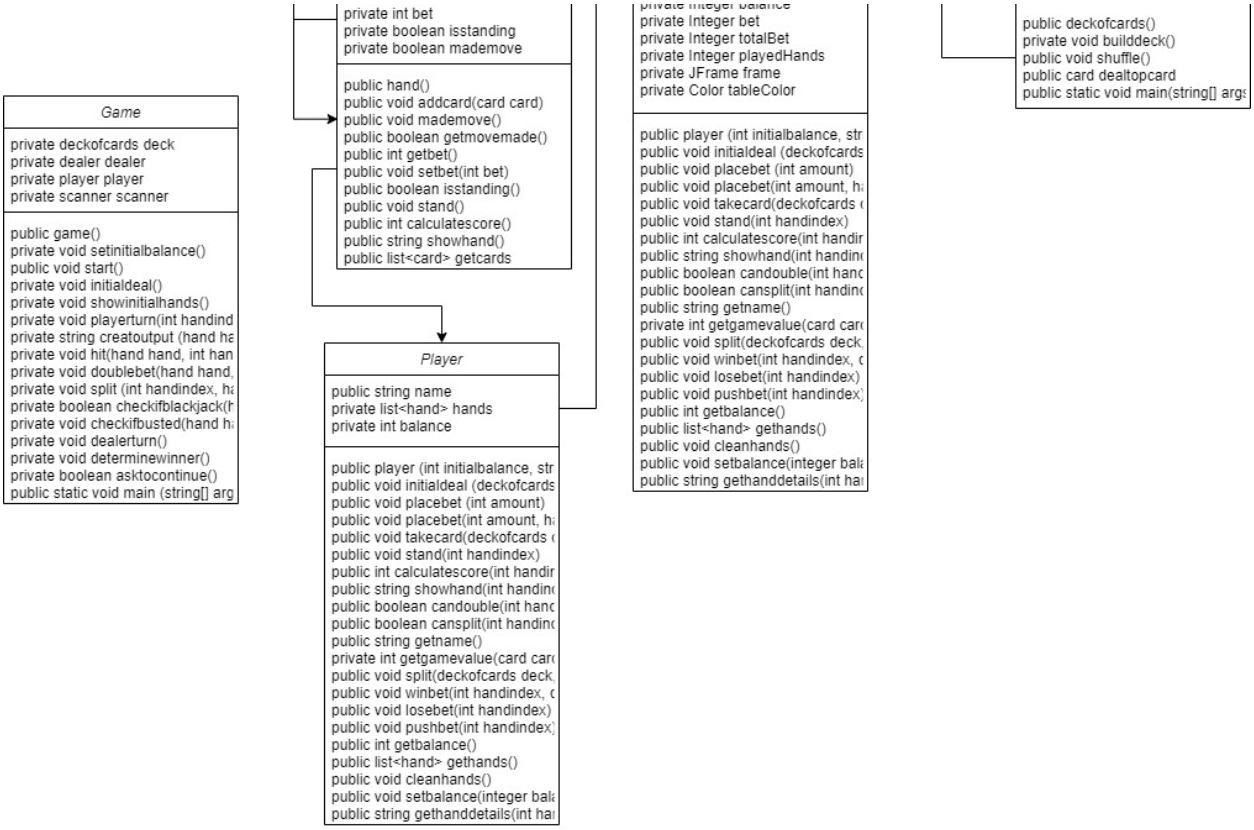
Major Components:

- Deck of cards: multiple decks, shuffling, deal off the top, deal in order.
    - Establish individual cards that hold score values that make a full 52-card deck. Create three of them and ensure they all shuffle together. Cards are dealt off the top of the deck, one by one, in order of the players and the dealer.
    - Cards are dealt in order beginning to the left of the dealer. The dealer must also collect bets, facilitate player bet choices, and pay out bets.
    - Players can input their name, input bet amounts, be dealt cards, play hands, replay, and save their progress.

Game/UI Features:

- Game table, display currency, action buttons, and outcome display.
    - A game table that holds all cards and action buttons on a grid that acts as a playing table.
    - Display card images, bet amounts, and dealer cards.
    - Outcome display that illustrates user action, bet winnings/losses, and user input to play again or quit.




Game
private deckofcards deck
private dealer dealer
private player player
private scanner scanner
public game()
private void setinitialbalance()
public void start()
private void initialdeal()
private void showinitialhands()
private void playerturn(int handind
private string creatoutput (hand ha
private void hit(hand hand, int han
private void doublebet(hand hand,
private void split (int handindex, ha
private boolean checkifblackjack(h
private void checkifbusted(hand ha
private void dealerturn()
private void determinewinner()
private boolean asktocontinue()
public static void main (string[] arg
private int bet
private boolean isstanding
private boolean mademove
public hand()
public void addcard(card card)
public void mademove()
public boolean getmovemade()
public int getbet()
public void setbet(int bet)
public boolean isstanding()
public void stand()
public int calculatescore()
public string showhand()
public list<card> getcards
Player
public string name
private list<hand> hands
private int balance
public player (int initialbalance, str
public void initialdeal (deckofcards
public void placebet (int amount)
public void placebet(int amount, ha
public void takecard(deckofcards
public void stand(int handindex)
public int calculatescore(int handir
public string showhand(int handin
public boolean candouble(int hanc
public boolean cansplit(int handin
public string getname()
private int getgamevalue(card car
public void split(deckofcards deck,
public void winbet(int handindex,
public void losebet(int handindex)
public void pushbet(int handindex)
public int getbalance()
public list<hand> gethands()
public void cleanhands()
public void setbalance(integer bal
public string gethanddetails(int ha
private Integer bet
private Integer totalBet
private Integer playedHands
private JFrame frame
private Color tableColor
public player (int initialbalance, str
public void initialdeal (deckofcards
public void placebet (int amount)
public void placebet(int amount, ha
public void takecard(deckofcards
public void stand(int handindex)
public int calculatescore(int handir
public string showhand(int handin
public boolean candouble(int hanc
public boolean cansplit(int handin
public string getname()
private int getgamevalue(card car
public void split(deckofcards deck,
public void winbet(int handindex,
public void losebet(int handindex)
public void pushbet(int handindex)
public int getbalance()
public list<hand> gethands()
public void cleanhands()
public void setbalance(integer bal
public string gethanddetails(int ha
public deckofcards()
private void builddeck()
public void shuffle()
public card dealtopcard
public static void main(string[] args

V. Test Plan

Most of the testing revolved around checking conditions. Checking the hand's possibility for betting options, checking the player's balance for sufficient funds, and checking that payouts are awarded accordingly (based on different betting options). Tests were also implemented to check for winners, losers, pushes, and Blackjack hands (that immediately win the hand and award 1.5x). Ignore the errors, these tests do in fact work- I just woke up and decided it was time to break my entire code before making this document. I digress. Once these tests returned pass, we implemented the designs into the main project. The code below checks for the false conditions required for splitting a hand: an invalid hand (non-pair) and insufficient funds to place a bet.

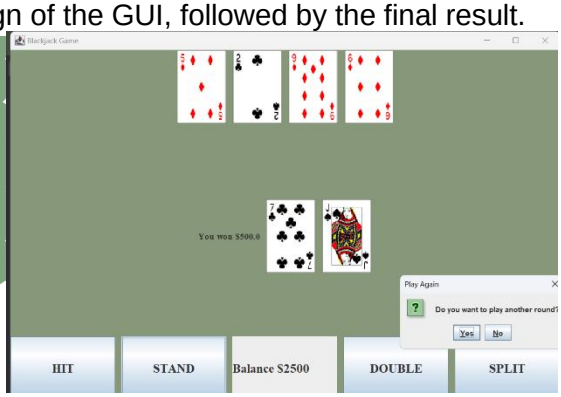
```
        // Assert
        assertTrue(canSplit);
    }

    @Test
    public void testCanSplit_InvalidHandIndex() {
        // Arrange
        int handIndex = 2; // Invalid hand index

        // Act
        boolean canSplit = player.canSplit(handIndex);

        // Assert
        assertFalse(canSplit);
    }

    @Test
    public void testCanSplit_InsufficientBalance() {
        // Arrange
        int handIndex = 0;
        hand1.setBet(200); // Set bet to 200
        player.setBalance(100); // Set balance to 100

        // Create a hand with two cards of the same rank
        Card card1 = new Card(Card.KING, Card.HEARTS);
        Card card2 = new Card(Card.KING, Card.DIAMONDS);
        hand1.addCard(card1);
        hand1.addCard(card2);

        // Act
        boolean canSplit = player.canSplit(handIndex);

        // Assert
        assertFalse(canSplit);
```

The code below tests for determining pushes, determining player Blackjack hand wins, and testing for when a player busts (exceeds 21).

```
    public void testDetermineWinner_Tie() {
        // Arrange
        playerHand1.addCard(new Card(Card.KING, Card.HEARTS));
        playerHand1.addCard(new Card(Card.QUEEN, Card.DIAMONDS));
        dealerHand.addCard(new Card(Card.TEN, Card.CLUBS));
        dealerHand.addCard(new Card(Card.TEN, Card.SPADES));

        // Act
        game.determineWinner();

        // Assert
        String expectedOutput = "Final scores for hand 1:\r\nDealer: 20\r\nPlayer: 20\r\nIt's a tie with hand 1\r\n";
        assertEquals(expectedOutput, outContent.toString());
    }

    @Test
    public void testDetermineWinner_PlayerBlackjack() {
        // Arrange
        playerHand1.addCard(new Card(Card.ACE, Card.HEARTS));
        playerHand1.addCard(new Card(Card.TEN, Card.DIAMONDS));
        dealerHand.addCard(new Card(Card.FIVE, Card.CLUBS));
        dealerHand.addCard(new Card(Card.SIX, Card.SPADES));

        // Act
        game.determineWinner();

        // Assert
        String expectedOutput = "Final scores for hand 1:\r\nDealer: 11\r\nPlayer: 21\r\nPlayer wins with hand 1\r\n";
        assertEquals(expectedOutput, outContent.toString());
    }

    @Test
    public void testDetermineWinner_PlayerBust() {
        // Arrange
        playerHand1.addCard(new Card(Card.KING, Card.HEARTS));
```

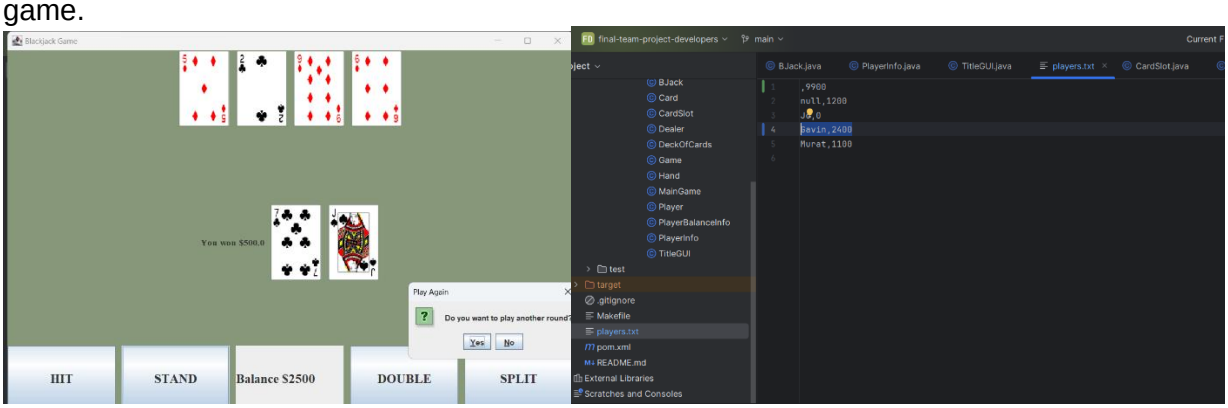
VI. Project Implementation Description

In the original prototype, we planned to have separate chips where players would bet from (or at least chip visualizations for bets placed), rather than a total dollar balance. Ultimately, the final product bets from dollar amounts, not chip equivalents. Furthermore, in the original prototype, we intended to include Insurance. Insurance is a side bet equal to the first available to players if the dealer shows an Ace card. Should the dealer flip a natural 21, the insurance pays the player 2-1, otherwise, the player simply loses the insurance bet. In the end, due to time constraints and technical issues, we did not include Insurance. We also intended for local multiplayer capability, however, were only able to implement single player and progress saving. Below will be the original design of the GUI, followed by the final result.



As shown, we planned for a much more detailed and colorful GUI. But, because of time constraints and failed testing, we were only able to implement the functional pieces and less of the visual design pieces.

Below, are a couple more screenshots from the game process. The first image shows the GUI in action while the second image shows the progress being saved, as result of the game.



VII. Future Work

Overall, I am very excited about the result. All the essential pieces work accordingly while only missing a few features. In the future, we would implement chips for currency, add insurance betting, and local multiplayer capacity. Furthermore, on the less functional end, updated the GUI with better graphics, more details, background music, action sounds, and maybe a dealer advice button- that would offer advice (Ex. “What would the dealer do with my hand?”). In completing these tasks, the functional pieces must be completed first. This ensures that the product remains usable at any stage of development. Once the game plays exactly as it's supposed to, update the balance to print as chips, rather than dollar amounts. Lastly, all the small quality of life updates can be addressed. Rather than a blank color for a table, add a wood border and texture. Display chips and moving items (simulating real gambling), overall giving it a more 3D feel.

VIII. Glossary

- **Bet:** Money wager placed by player.
- **Hit:** Dealt another card.
- **Stand:** To stay with the cards dealt and end their turn.
- **Double:** On the original two cards dealt, place a bet equal to the original and you are dealt a single card before the turn ends.
- **Split:** Holding a pair can be split into two hands and must be met with a bet from the player. You now have two hands and two bets to play.
- **Insurance:** Side bet offered to the player if the dealer’s up-card is an Ace, as insurance against the dealer’s hand being a Blackjack. Insurance pays out 2-1.
- **Blackjack:** When the player or dealer flips a natural 21. Blackjack pays out 1.5x
- **Push:** A tie between player and dealer, where money is “pushed” (did not win or lose).
- **Class:** The template from which objects are created.
- **Method:** Block of code that, when called, performs specific actions mentioned in it.
- **GUI:** Graphical User Interface. A user-friendly visual experience builder for Java applications.
- **Action:** Any act caused by the user that produces an outcome within the program

- **Action Button:** GUI options that can be clicked to cause events within the program.

IX. References

X. Appendices