VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING



**Data Structure and Algorithm**

**IT013IU**

FINAL REPORT

MSc: Thai Trung Tin

**TOPIC: OVER-UNDER BETTING**

#### BY GROUP DSA– MEMBER LIST

|  |  |  |
| --- | --- | --- |
| NGUYEN DU NHAN | ITDSIU22140 | Leader |
| NGUYEN THE HAO | ITDSIU22139 | Member |

**TABLE OF CONTENTS**

[ABSTRACT 2](#_Toc136791164)

[CHAPTER 1: INTRODUCTION 3](#_Toc1458714990)

[1.1 Overview of Over-Under Betting 4](#_Toc2009789450)

[1.2 Developer team 4](#_Toc1905418998)

[CHAPTER 2: SOFTWARE REQUIREMENTS 4](#_Toc2120776733)

[2.1 Development Environment 5](#_Toc1865823086)

[2.2 Functional Requirements 5](#_Toc1666402629)

[2.3 Non-Functional Requirements 6](#_Toc1232182577)

[CHAPTER 3: DESIGN 6](#_Toc1340061496)

[3.1 Overview 7](#_Toc2041423766)

[3.2 UML 7](#_Toc229381996)

[CHAPTER 4: IMPLEMENTATION 8](#_Toc57614825)

[4.1 uitls 8](#_Toc2008433815)

[a. Utils.java 8](#_Toc1509961557)

[b. Simulator.java 11](#_Toc768128469)

[4.2 model 13](#_Toc125763438)

[a. Player.java 14](#_Toc1538442757)

[b. House.java 16](#_Toc1488227674)

[4.3 bias 17](#_Toc1182288730)

[a. BiasManager.java 17](#_Toc1299357883)

[4.4 stats 29](#_Toc844658621)

[a. StatisticsManager.java 29](#_Toc1635463298)

[4.5 engine 31](#_Toc1488081880)

[a. GameEngine.java 31](#_Toc1649387650)

[4.6 resources 34](#_Toc1931317413)

[4.7 panel 36](#_Toc136320498)

[a. HomePanel.java 36](#_Toc1346375724)

[b. GamePanel.java 39](#_Toc570343438)

[c. ResultBar.java 42](#_Toc303087928)

[d. MainFrame.java 44](#_Toc2109654673)

[4.8 Result 46](#_Toc1176419713)

[CHAPTER 5: FINAL APP GAME 53](#_Toc784297708)

[5.1 Source code (link github): 54](#_Toc1701360875)

[5.2 Demo video: 54](#_Toc1957912237)

[5.3 Instruction: 54](#_Toc322968484)

[a. Main Menu 54](#_Toc1979951529)

[b. Game Panel 54](#_Toc149392505)

[c. Game Play 55](#_Toc78737659)

[CHAPTER 6: EXPERIENCE 62](#_Toc473880072)

# **ABSTRACT**

This project develops a simulation-based dice betting game featuring an intelligent bias system to mimic real-world gambling dynamics. The system adjusts dice outcomes based on player behavior, betting patterns, and balance progression using statistical tracking and conditional probability. A Bias Manager component strategically enforces house wins while preserving the illusion of fairness, particularly countering Martingale-like strategies. Additionally, a simulator module tests long-term outcomes against pattern-driven players. Results demonstrate that adaptive biasing can effectively manage risk and sustain profitability, offering insights into behavioral modeling and strategy resistance in probabilistic games.

# **CHAPTER 1: INTRODUCTION**

## **1.1 Overview of Over-Under Betting**

The project implements a digital dice betting game where players wager on whether the total sum of three six-sided dice will fall in the "under" (3–10) or "over" (11–18) range. The game interface is built using Java Swing, offering an interactive and visually engaging experience. Players input their name, select a starting balance, and place bets via predefined buttons or custom amounts. Core gameplay consists of betting rounds where the outcome is determined by simulated dice rolls. To enhance realism, a result bar visualizes recent game outcomes, and balance updates are displayed dynamically. The game supports win/loss tracking and strategic gameplay elements.

## **Developer team**

We have 2 members team members from International University:

|  |  |  |
| --- | --- | --- |
| Name - GitHub username | ID | Contribute |
| Nguyen Du Nhan - [DuNhan1930](https://github.com/DuNhan1930) | ITDSIU22140 | Write report  Design program  Fix Bug |
| Nguyen The Hao – [THao0247](https://github.com/THao0247) | ITDSIU22139 | Slide  Design GUI  Fix Bug |

# 

# **CHAPTER 2: SOFTWARE REQUIREMENTS**

## **2.1 Development Environment**

**Java Development Kit (JDK):** The game is developed in Java, requiring JDK 8 for compiling and executing the program. Java offers strong object-oriented features and cross-platform compatibility, making it ideal for developing GUI-based desktop applications.

**Integrated Development Environment (IDE):** IntelliJ IDEA is used as the primary development tool due to its intelligent code assistance, debugging features, and seamless integration with Java Swing and version control systems. Its project structure and build tools enhance code maintainability.

**Graphical User Interface Library:** The game leverages Java Swing, a built-in Java GUI framework, for rendering windows, panels, buttons, and custom components such as dice icons and the result bar. Swing’s layout managers and painting utilities are used extensively.

## **2.2 Functional Requirements**

**Player Setup and Input:** Players must be able to input their name and starting balance, which initializes their session. This input is validated to ensure correct formatting and values.

**Betting Mechanics:** Players choose between two options—"Under (3–10)" or "Over (11–18)"—and specify a bet amount via text input or predefined bet buttons (e.g., 5$, 10$, All In). Invalid or excessive bets are rejected with user-friendly messages.

**Dice Rolling and Animation:** Each round triggers an animated dice roll sequence followed by the result display using image icons. The result is determined either fairly or through an adaptive bias mechanism.

**Bias Management System:** The core game logic uses a BiasManager to simulate a smart house strategy. It analyzes the player's betting patterns, win/loss streaks, and amount bet to adjust dice outcomes probabilistically in the house’s favor.

**Balance and Result Tracking:** Player balance updates after each round, with winnings or losses reflected immediately. A visual result bar tracks the last 15 dice outcomes, displaying them as colored markers with totals.

**Win Streak Display:** The interface displays the current win streak beside the balance, providing motivational feedback and encouraging continued play.

**Game Over Condition:** When the player's balance reaches zero, the game disables further input and notifies the user with a terminal message.

**Simulation Module:** A separate simulator is included to test the bias engine against automated betting strategies over tens of thousands of rounds, supporting empirical evaluation of win rates and house advantage.

## **2.3 Non-Functional Requirements**

**Performance:** The game must respond to user actions—such as placing bets, clicking buttons, or animations—within 200 milliseconds. Simulations should complete 100,000 rounds in under 10 seconds on a modern CPU.

**Usability:** The interface must be clear and intuitive. Font choices, button sizes, hover effects, and confirmation dialogs should promote ease of interaction for users of varying experience levels.

**Visual Design:** Dice icons, result bars, labels, and layout panels should follow a consistent visual theme. Animated transitions and gradient backgrounds enhance immersion without sacrificing clarity.

**Reliability and Robustness:** The game must handle incorrect input gracefully, avoiding crashes or logic errors. Simulation mode should be isolated from the interactive mode to prevent state interference.

# **CHAPTER 3: DESIGN**

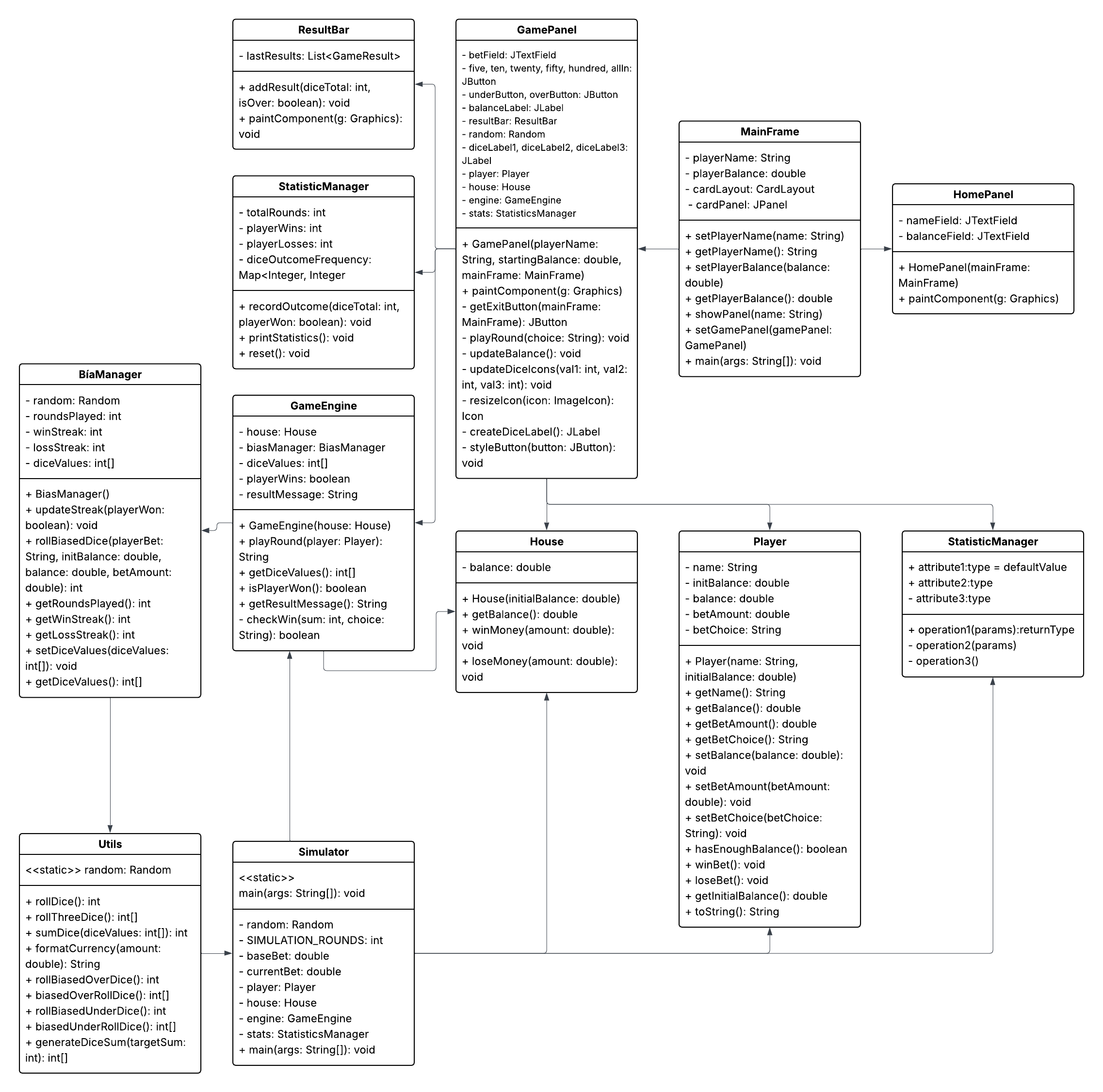
## **3.1 Overview**

This project is designed using a modular, object-oriented architecture. The system is divided into clearly defined packages, each responsible for a specific domain of functionality.

* **utils:** Contains utility functions (Utils.java) and a simulation tool (Simulator.java) used for testing bias behavior statistically.
* **model:** Defines data classes for the Player and the House, tracking balance and betting details.
* **bias:** Implements the core intelligent bias system in BiasManager.java, which adapts dice outcomes based on betting strategy patterns.
* **stats:** Offers data collection and analytics via StatisticsManager.java, helping evaluate system fairness and player performance.
* **engine:** The game logic core (GameEngine.java) orchestrates gameplay, invokes bias logic, and updates game state.
* **resources:** Contains media assets like dice images for the UI.
* **panel:** Manages the GUI components (HomePanel, GamePanel, ResultBar, MainFrame) that interact with the user and visualize game events.

This layered design enhances clarity, supports maintainability, and isolates critical logic for testing and extension. Further implementation specifics will be detailed in the following chapter.

## **3.2 UML**



# **CHAPTER 4: IMPLEMENTATION**

## **4.1 uitls**

### **a. Utils.java**

The *Utils* class provides foundational utility methods essential for randomization, dice simulation, formatting, and mathematical operations. These are critical in enabling consistent and reusable logic throughout the bias engine and game simulation.

**Core Functionalities**

1. **randomChance(double probability)**

|  |
| --- |
| public static boolean randomChance(double probability) {  return Math.*random*() < probability; } |

* **Purpose**: Returns true with a probability p, simulating probabilistic decisions.
* **Complexity**: **O(1)** constant time, using Math.random() internally.
* **Use Case**: Supports probability-based bias control in the game logic.

1. **randomBetween(int min, int max)**

|  |
| --- |
| public static int randomBetween(int min, int max) {  return *random*.nextInt(max - min + 1) + min; } |

* **Purpose**: Generates a random integer between min and max (inclusive).
* **Complexity**: **O(1)**.
* **Use Case**: Used frequently for randomizing dice outcomes, sums, or bias-based conditions.

1. **rollDice() and rollThreeDice()**

|  |
| --- |
| public static int rollDice() {  return *random*.nextInt(6) + 1; }  public static int[] rollThreeDice() {  int[] diceValues = new int[3];  diceValues[0] = *rollDice*();  diceValues[1] = *rollDice*();  diceValues[2] = *rollDice*();  return diceValues; } |

* **Purpose**: Simulates fair dice rolls. The latter returns an array of three integers, each in range 1–6.
* **Complexity**: rollDice() is **O(1)**, rollThreeDice() is **O(1)** due to fixed-size array operations.
* **Use Case**: Core to the random and biased dice rolling systems.

1. **sumDice(int[] diceValues)**

|  |
| --- |
| public static int sumDice(int[] diceValues) {  int sum = 0;  for (int value : diceValues) {  sum += value;  }  return sum; } |

* **Purpose**: Returns the total of three dice values.
* **Complexity**: **O(n)**, where n is the size of the input array. Since n = 3 is fixed, it's effectively constant time.
* **Use Case**: Required for outcome evaluation and scoring systems.

1. **formatCurrency(double amount)**

|  |
| --- |
| public static String formatCurrency(double amount) {  return String.*format*("$%.2f", amount); } |

* **Purpose**: Formats a double into a standard dollar string format.
* **Complexity**: **O(1)**, as string formatting is fixed-cost.
* **Use Case**: Enhances readability of balances during gameplay and statistics reporting.

1. **generateDiceSum(int targetSum)**

|  |
| --- |
| public static int[] generateDiceSum(int targetSum) {  int[] dice = new int[3];  do {  dice[0] = *rollDice*();  dice[1] = *rollDice*();  dice[2] = targetSum - dice[0] - dice[1];  } while (dice[2] < 1 || dice[2] > 6);  return dice; } |

* **Purpose**: Generates a 3-element array of dice values whose total equals targetSum, maintaining valid dice values (1–6).
* **Algorithm**:
* Randomly assign values to the first two dice.
* Calculate the third as the difference from targetSum.
* Retry (loop) until all values are within [1, 6].
* **Complexity**: **Expected O(1)** – Since the number of valid combinations is large within dice constraints, the loop converges quickly.
* **Use Case**: Core mechanism to enforce biased results in a controllable and statistically realistic way.

**Benefit**

* **Stateless and Reusable**: All methods are static and avoid side effects, enabling widespread use without instantiation.
* **Optimized for Small Data**: Given fixed-size structures (arrays of size 3), time and space complexity remain minimal.
* **Extensible**: The class can easily be expanded to support future needs such as weighted probabilities or advanced simulations.

### **b. Simulator.java**

The *Simulator* class is a utility designed to evaluate the performance of the game engine and bias logic over large-scale test runs. It mimics an intelligent player (SmartBot) applying a simple predictive strategy, allowing the system to measure fairness, balance, and robustness against pattern-based exploitation.

**Core Functionalities**

1. **Simulation Loop**

|  |
| --- |
| static final int *SIMULATION\_ROUNDS* = 100000;  if (player.getBalance() < *BASE\_BET*) break; |

* Executes up to *100,000* rounds or until the player runs out of balance.
* Time Complexity: O(N), where N = number of rounds (bounded by constant *SIMULATION\_ROUNDS*).

1. **Backtracking Behavior Modeling**

|  |
| --- |
| Deque<Boolean> lastWins = new ArrayDeque<>();  Deque<Integer> lastSums = new ArrayDeque<>();  if (lastWins.size() == 5) lastWins.removeFirst();  if (lastSums.size() == 5) lastSums.removeFirst();  lastWins.add(playerWon);  lastSums.add(total); |

* Uses two sliding windows (of size 5):
* *Deque<Boolean> lastWins*: track win/loss history.
* *Deque<Integer> lastSums*: track previous dice totals.
* Based on the counts of "over" vs. "under" in the past 5 rounds, the bot decides its next guess using a heuristic (trend-based prediction).
* Data Structure: *ArrayDeque* for O(1) insertion/removal at both ends.

1. **Martingale Strategy**

|  |
| --- |
| double betAmount = BASE\_BET;  if (!lastWins.isEmpty() && !lastWins.peekLast()) {  betAmount \*= 2;  } |

* If the last round was a loss, the bot doubles its bet.
* This simulates a common risk-amplifying betting tactic aimed at recouping prior losses.
* This behavior stresses the bias engine's ability to handle and penalize reckless betting.

1. **Adaptive Decision Making**

|  |
| --- |
| String guess;  if (lastWins.size() >= 5) {  guess = (under > over) ? "over" : "under";  } else {  guess = random.nextBoolean() ? "under" : "over";  } |

* If the sample size is insufficient (less than 5 rounds), the bot makes a random choice to simulate uncertainty.
* Once enough data is accumulated, a deterministic pattern recognition strategy is applied.

**Key Algorithmic Components**

1. **Trend Prediction Heuristic**

|  |
| --- |
| guess = (under > over) ? "over" : "under"; |

* Uses majority voting based on recent game outcomes.
* A simple yet effective O(k) sliding window analysis (k = 5).

1. **Bet Adjustment Logic**

|  |
| --- |
| if (!lastWins.isEmpty() && !lastWins.peekLast()) {  betAmount \*= 2;  } |

* Enforces exponential growth (martingale) only on loss, encouraging fast balance swings.

1. **Efficiency & Scalability**

* Memory usage is minimal: all queues are fixed-size (<= 5 elements).
* No deep recursion or heavy computation; can scale to millions of rounds if needed.

**Purpose and Benefits**

* **Stress Testing:** Measures how well the BiasManager responds to pattern exploitation and high-risk betting strategies.
* **Balance Checking:** Evaluates win/loss ratio, outcome fairness, and house profitability.
* **Data Generation:** Produces statistically significant results for tuning bias parameters.

## **4.2 model**

### **a. Player.java**

The *Player* class encapsulates all attributes and methods associated with a player’s betting behavior. It models key data such as identity, balance, betting decisions, and supports transactional methods for game outcomes. It acts as the primary agent interacting with the game engine and is essential in both gameplay and simulation scenarios.

**Core Data Structure**

|  |
| --- |
| private final String name;  private final double initialBalance;  private double balance;  private double betAmount;  private String betChoice; |

* **name:** Immutable identifier of the player.
* **initialBalance:** Snapshot used for tracking profit/loss overtime.
* **balance:** Mutable field reflecting real-time funds.
* **betAmount:** Current round’s wager.
* **betChoice:** "over" or "under" selected by the player.

**Core Functionalities**

1. **Balance Verification**

|  |
| --- |
| public boolean hasEnoughBalance() {  return balance >= betAmount;  } |

* Confirms player affordability for a given round.
* Used to halt rounds if balance falls below bet.

1. **Win/Loss Accounting**

|  |
| --- |
| public void winBet() {  balance += betAmount;  }  public void loseBet() {  balance -= betAmount;  } |

* Adjust balance accordingly.
* Ensures isolation of financial logic from the game engine.
* Facilitates accurate tracking in simulations and real games.

1. **Bet Configuration**

|  |
| --- |
| public void setBetAmount(double betAmount) {  this.betAmount = betAmount;  }  public void setBetChoice(String betChoice) {  this.betChoice = betChoice;  } |

* Allow the player (or AI) to set intention per round.
* This modular input design supports flexible player strategies (human, random, AI).

1. **Read-Only Interfaces**

|  |
| --- |
| public double getInitialBalance() {  return initBalance;  }  public double getBalance() {  return balance;  }  public double getBetAmount() {  return betAmount;  }  public String getBetChoice() {  return betChoice;  } |

* Used by game engine and statistics modules to retrieve player state.
* Ensures data encapsulation and immutability where appropriate.

**Algorithm**

* O(1) operations for all methods, designed for high-frequency invocation (e.g., 100,000 rounds in simulation).

**Benefits**

* **Encapsulation:** Clear boundary between data and logic.
* **Modularity:** Works seamlessly in both UI and CLI simulations.
* **Scalability:** Lightweight model supports millions of rounds without performance degradation.

### **b. House.java**

The House class models the financial state of the system that plays against the player - effectively representing the casino or host. It acts as the economic counterpart to the Player class and is responsible for adjusting internal balance based on game outcomes.

**Core Data Structure**

|  |
| --- |
| private double balance; |

* Maintains a single mutable scalar field.
* Represents total available funds of the house (i.e., the opponent of the player).
* The simplicity ensures deterministic performance and minimal memory use.

**Core Functionalities**

**Initialization**

|  |
| --- |
| public House(double initialBalance) {  this.balance = initialBalance; } |

* Constructs a House object with a given starting balance.

**Get Balance**

|  |
| --- |
| public double getBalance() {  return balance; } |

* Provides read-only access to the current house balance.
* Time complexity: O(1)

**Transactional Methods**

|  |
| --- |
| public void winMoney(double amount) {  this.balance += amount; } public void loseMoney(double amount) {  this.balance -= amount; } |

* Adjusts internal balance when player wins or loses.
* Keeps house consistent with player outcomes.
* No validation needed here - correctness is enforced externally (e.g., in GameEngine).
* Time complexity: O(1)

**Algorithm**

* All operations have O(1) time and space complexity.
* No branching, loops, or nested structures - deterministic and thread-safe (under single-threaded execution).
* Memory footprint is minimal (single primitive variable).

**Benefits**

* **Simplicity & Efficiency:** Pure value-holder with atomic financial operations.
* **Scalability:** Can sustain millions of updates with no slowdown.
* **Transparency:** Clear abstraction of game bank logic - no mixing with game rules or player behavior.

## **4.3 bias**

### **a. BiasManager.java**

BiasManager acts as an intelligent strategy controller for manipulating dice rolls. Its role is to:

* Simulate a biased game environment
* Analyze player behavior
* Dynamically adjust bias logic to counteract exploitable strategies (e.g. Martingale, repetition)
* Maintain a balance between perceived fairness and house advantage

**Core Data Structures**

|  |
| --- |
| private int roundsPlayed = 0; private int winStreak = 0; private int lossStreak = 0; private int[] diceValues = new int[3];  private final LinkedList<Boolean> recentOvers = new LinkedList<>(); private final int memorySize = 6;  // Track player betting behavior private final LinkedList<String> recentChoices = new LinkedList<>(); private final LinkedList<Double> recentBets = new LinkedList<>(); private final LinkedList<Boolean> recentWins = new LinkedList<>();  private static final int *MAX\_HISTORY* = 5; private int maxConsecutiveBiasLosses = 0; private static final int *MAX\_BIAS\_LOSSES* = 4; |

* **private int roundsPlayed;:** Tracks the number of rounds executed. Helps adjust time-dependent bias probability.
* **private int winStreak;:** Tracks consecutive rounds the player has won. Used to decide when to trigger bias if the player is on a winning streak.
* **private int lossStreak;:** Tracks consecutive rounds the player has lost. Similar to winStreak, it contributes to adaptive bias decisions based on the player’s losing streak.
* **private int[] diceValues = new int[3];:** Stores the outcome of the current dice roll. Used to record and return the rolled values.
* **private final LinkedList<Boolean> recentOvers;:** Maintains a sliding window indicating whether recent dice sums were “over” (>10). Helps prevent repetitive patterns in game outcomes.
* **private final LinkedList<String> recentChoices;:** Stores the player’s most recent bet choices ("over"/"under"). Detects repetitive player behavior.
* **private final LinkedList<Double> recentBets;:** Records recent bet amounts. Enables detection of Martingale strategies by analyzing bet progression.
* **private final LinkedList<Boolean> recentWins;:** Tracks whether the player won each recent round. Helps determine if the player is using riskier strategies after losses.
* **private static final int MAX\_HISTORY = 5;:** Defines the maximum size of the sliding history windows. Ensures behavioral tracking remains recent and lightweight.
* **private int maxConsecutiveBiasLosses;:** Counts how many times the bias engine has failed consecutively. Prevents overuse of bias when it becomes ineffective.
* **private static final int MAX\_BIAS\_LOSSES = 4;:** Threshold to temporarily disable bias when it fails repeatedly. Prevents over-biasing and reduces the chance of detection by players.

**Core Algorithmic Components**

1. **Streak Tracking**

|  |
| --- |
| public void updateStreak(boolean playerWon) {  roundsPlayed++;  if (playerWon) {  winStreak++;  lossStreak = 0;  maxConsecutiveBiasLosses = 0;  } else {  lossStreak++;  winStreak = 0;  }  } |

This method is responsible for updating the internal state of the system based on whether the player won the last round. It tracks:

* Total number of rounds played (*roundsPlayed*)
* Ongoing winning streak (*winStreak*)
* Ongoing losing streak (*lossStreak*)
* Resets bias failure counter (*maxConsecutiveBiasLosses*) when a win occurs

These streaks influence biasing decisions elsewhere in the logic. For example:

* Long winning streaks may trigger stronger bias to break it.

Long losing streaks combined with rising bets may also be flagged as high-risk requiring bias.

**Algorithm**

*roundsPlayed++*: Increments the total count of rounds (used to increase bias probability over time).

If *playerWon == true*:

* *winStreak++* continues the winning streak.
* *lossStreak = 0* resets losing counter.
* *maxConsecutiveBiasLosses = 0* resets safety threshold.

If *playerWon == false*:

* *lossStreak++* increments loss streak.
* *winStreak = 0* clears win streak (only one streak is active at a time).

**Time Complexity:**

All operations are O(1) - constant time, single comparisons and assignments.

**Usage Context**

**Downstream Use**: These counters are used by:

* shouldBias(...) to help decide whether the house should intervene.
* rollBiasedDice(...) indirectly depends on how streaks evolve.

**Risk Management**: Tracks if the player is exploiting the game or is on a “lucky” trend, signaling when to increase or reduce bias accordingly.

1. **Biased Dice Rolling**

|  |
| --- |
| public int rollBiasedDice(String playerBet, double initBalance, double balance, double betAmount) {  roundsPlayed++;  updateBehaviorHistory(playerBet, betAmount);  boolean doBias = shouldBias(playerBet, initBalance, balance, betAmount);  if (maxConsecutiveBiasLosses >= MAX\_BIAS\_LOSSES) {  doBias = false;  maxConsecutiveBiasLosses = 0;  }  boolean avoidPattern = isOverRepeatedTooMuch();  if (betAmount <= 0.1 \* balance && Utils.randomChance(0.25)) {  useNaturalWin(playerBet);  return Utils.sumDice(diceValues);  }  if (doBias && !avoidPattern) {  if (Utils.randomChance(0.8)) {  forceLoss(playerBet);  } else {  int biasedSum = playerBet.equalsIgnoreCase("over")  ? Utils.randomBetween(6, 10)  : Utils.randomBetween(11, 14);  diceValues = Utils.generateDiceSum(biasedSum);  recordOverResult(biasedSum > 10);  maxConsecutiveBiasLosses++;  }  } else {  diceValues = Utils.rollThreeDice();  recordOverResult(Utils.sumDice(diceValues) > 10);  }  return Utils.sumDice(diceValues);  }  private void forceLoss(String playerBet) {  int biasedSum = playerBet.equalsIgnoreCase("over")   ? Utils.*randomBetween*(3, 10)   : Utils.*randomBetween*(11, 18);  diceValues = Utils.*generateDiceSum*(biasedSum);  recordOverResult(biasedSum > 10);  maxConsecutiveBiasLosses++; } |

This is the central algorithm that determines whether to intervene in a game round using bias. It simulates how the house may try to reduce player wins based on risk analysis, past betting patterns, and randomness.

**Algorithm**

**Increment Rounds**:

* roundsPlayed++: tracks the progress of time to enable time-based bias.

**Track Player Behavior**:

* updateBehaviorHistory(...) saves the player's current bet and amount into fixed-size history queues (sliding window).

**Decide Bias**:

* shouldBias(...) returns a boolean depending on:
* Player profit level
* Bet aggressiveness
* Repeated patterns (e.g. same choice repeatedly)
* Use of martingale strategy

**Pattern Suppression**:

* If bias was too strong recently (4+ failed bias rounds), bias is disabled temporarily.
* isOverRepeatedTooMuch() prevents suspiciously repeated “over” / “under” results.

**Natural Win for Small Bet**:

* If the bet is very small, allow the player to win 25% of the time to look fair.

**Execute Bias**:

* 80% of the time, the house forces a loss: forceLoss(...), which will generate a total randomly opposite with player’s choice.
* 20%: apply a mild bias by forcing a total close to the loss boundary (e.g. 10 or 11).

**Fair Roll Otherwise**:

* Uses Utils.rollThreeDice() for unbiased, fair behavior.

**Time Complexity**:

* Dominated by fixed-size operations: history sliding windows are size ≤ 5 → **O(1)**
* The only loop (Utils.generateDiceSum(...)) is bounded with retry-until-valid logic, still O(1) in expected time.

**Modular Interaction**

* **Helper Methods:** *shouldBias(...)*, *forceLoss(...)*, *useNaturalWin(...)*, *isOverRepeatedTooMuch()* keep this method clean and modular.
* **Integration:** Called once per round from GameEngine.playRound().

1. **Bias Decision Logic**

|  |
| --- |
| private boolean shouldBias(String playerBet, double initBalance, double balance, double betAmount) {  double profit = balance / initBalance;  double timeBiasRate = Math.*min*(0.3, roundsPlayed \* 0.01);  double ratioBet = betAmount / balance;   boolean isMartingale = isIncreasingBetAfterLoss();  boolean isRepeatingChoice = isChoiceBiased(playerBet);   if (profit >= 2) return true;  if (isMartingale) return true;  if (isRepeatingChoice && Utils.*randomChance*(0.7)) return true;  if (lossStreak > 3 || winStreak > 3) return ratioBet > 0.4;  return Utils.*randomChance*(timeBiasRate); } |

This function contains biased decision logic. It evaluates multiple risk-related conditions to determine whether the house should intervene and bias the dice outcome against the player.

**Algorithm:**

**Profit Check**:

|  |
| --- |
| if (profit >= 2) return true; |

* If the player has doubled their money (≥ 2× initial balance), the house becomes aggressive by always applying bias.
* This simulates a profit cap protection mechanism.

**Martingale Strategy Detection**:

|  |
| --- |
| if (isMartingale) return true; |

* If the player is doubling bets after a loss, bias activates to counter high-risk betting patterns.
* Detected via *isIncreasingBetAfterLoss()* (covered in a later step).

**Pattern Detection in Choice**:

|  |
| --- |
| if (isRepeatingChoice && Utils.randomChance(0.7)) return true; |

* If the player is repeatedly choosing the same bet (e.g., always "over"), bias may be applied with 70% chance to avoid predictability.

**Streak-Based Trigger**:

|  |
| --- |
| if (lossStreak > 3 || winStreak > 3) return ratioBet > 0.4; |

* If the player or the house is on a streak and the bet size is large (over 40% of balance), bias kicks in.
* This uses past streak tracking to balance risk.

**Time-Based Biasing**:

|  |
| --- |
| return Utils.randomChance(timeBiasRate); |

* A fallback soft-bias that grows gradually over time with gameplay.
* Encourages randomness in early rounds and lessens early detection by players.

**Time Complexity:**

* Each condition is a simple arithmetic or boolean check → O(1)
* Helper functions *isIncreasingBetAfterLoss()* and *isChoiceBiased(...)* also run in *O(k)*, where k = *MAX\_HISTORY* (constant) ⇒ effectively O(1).

1. **Behavior Pattern Detection**

This part of the system aims to detect strategic betting behavior - particularly Martingale (doubling after loss) and repetitive betting patterns - so the house can bias dice outcomes more effectively.

**Martingale Detection:**

|  |
| --- |
| private boolean isIncreasingBetAfterLoss() {  if (recentBets.size() < 2 || recentWins.size() < 2) return false;  double last = recentBets.getLast();  double prev = recentBets.get(recentBets.size() - 2);  boolean lastLost = !recentWins.get(recentWins.size() - 2);  return last > prev && lastLost; } |

**Algoritm:**

1. Ensure there’s enough history (at least 2 rounds).
2. Compare current bet to previous one.
3. Check if the second-last round was a loss.
4. Return true only if the bet increased after a loss.

**Time complexity:** O(1) - simple comparisons and retrieval from LinkedList.

**Choice Repetition Detection**

|  |
| --- |
| private boolean isChoiceBiased(String current) {  long count = recentChoices.stream()  .filter(choice -> choice.equalsIgnoreCase(current))  .count();  return count >= *MAX\_HISTORY* - 1; } |

**Algorithm:**

1. Traverse the sliding window of recent choices.
2. Count how many times the player bet the same as their current choice.
3. If it’s 4 out of 5 (*MAX\_HISTORY − 1*), it's considered repetitive.

**Time Complexity:** O(k), where k = *MAX\_HISTORY* (5) → effectively O(1).

1. **Behavior History Management**

|  |
| --- |
| private void updateBehaviorHistory(String choice, double betAmount) {  if (recentChoices.size() >= *MAX\_HISTORY*) recentChoices.removeFirst();  if (recentBets.size() >= *MAX\_HISTORY*) recentBets.removeFirst();  recentChoices.add(choice);  recentBets.add(betAmount); } |

Maintains a fixed-length record of the player's recent decisions and bet amounts for analysis by:

* *isChoiceBiased(...)*
* *isIncreasingBetAfterLoss()*

This history provides temporal context for adaptive biasing.

**Algorithm:**

1. Capacity control: If the queue has reached *MAX\_HISTORY* (i.e. 5), the oldest record is removed (*removeFirst()*).
2. Append new data:

* *recentChoices.add(choice)* → Tracks latest “over” / “under”.
* *recentBets.add(betAmount)* → Stores latest bet size.

The system ensures only the most recent k=5 actions are kept.

**Time complexity:**

* *removeFirst()*: O(1)
* *add()*: O(1)
* Total (per round): O(1)

1. **Round Outcome Tracking**

|  |
| --- |
| public void updateLastRoundResult(boolean playerWon) {  if (recentWins.size() >= *MAX\_HISTORY*) recentWins.removeFirst();  recentWins.add(playerWon); } |

Maintains a fixed-size window of recent round results:

* **true** if the player won the round.
* **false** if the player lost.

This is used primarily by the method *isIncreasingBetAfterLoss()* to detect Martingale-style strategies (bet doubling after loss).

**Algorithm:**

1. **Window maintenance:** When the list is full (*MAX\_HISTORY* = 5), discard the oldest result.
2. **Update:** Add the outcome of the current round to the tail.

This mirrors a FIFO queue, preserving the latest results in temporal order.

**Time complexity:**

* *removeFirst()*: O(1)
* *add()*: O(1)
* Total (per round): O(1)

1. **Anti-Pattern Defense**

|  |
| --- |
| private void recordOverResult(boolean isOver) {  if (recentOvers.size() >= memorySize) recentOvers.removeFirst();  recentOvers.add(isOver); } private boolean isOverRepeatedTooMuch() {  if (recentOvers.size() < memorySize) return false;  long count = recentOvers.stream().filter(b -> b).count();  return count >= memorySize - 1 || count <= 1; } |

* *recordOverResult*: Records whether the sum of the dice in the current round was “over” (sum > 10).
* *isOverRepeatedTooMuch*: Prevents long unbroken sequences of “over” or “under” outcomes, which might indicate or create bias.

These methods form a pattern regularize to ensure a realistic and fair-looking outcome distribution.

**Algorithm:**

1. *recordOverResult(boolean isOver)*

* Appends the boolean result of the dice sum:
* *true* for over (sum > 10).
* *false* for under (sum ≤ 10).
* Keeps the list at a fixed length (*memorySize = 6*).

Data Structure: LinkedList<Boolean> - supports O(1) insertions/removals.

1. *isOverRepeatedTooMuch()*

* Counts how many times "over" occurred in the recent window.
* Returns true if:
* Over occurred almost all the time (*≥ memorySize - 1*), or
* Rarely occurred (≤ 1).

This heuristic avoids generating statistically suspicious output like:

* *Over, Over, Over, Over, Over, Over*

**Time complexity:**

* *removeFirst()*: O(1)
* *add()*: O(1)
* *stream().filter().count()*: O(N), N = memorySize (small constant =6)
* Total (per round): O(1)

1. **Natural Win Granting**

|  |
| --- |
| private void useNaturalWin(String playerBet) {  int biasedSum = playerBet.equalsIgnoreCase("over")   ? Utils.*randomBetween*(11, 18)   : Utils.*randomBetween*(3, 10);  diceValues = Utils.*generateDiceSum*(biasedSum);  recordOverResult(biasedSum > 10); } |

This method manually generates a dice sum that lets the player win under specific conditions:

* The player’s bet is very small (typically ≤ 10% of balance).
* A probabilistic condition (e.g., 25%) is satisfied.

The goal is to appear generous, reduce suspicion, and allow occasional wins that do not financially hurt the house.

**Algorithm**

1. Determine Target Outcome:

* If the player bets "over" → generate a sum between 11–18.
* If the player bets "under" → generate a sum between 3–10.

1. Use utility method *Utils.generateDiceSum(targetSum)* to fabricate a realistic-looking dice combination that adds to the target sum.
2. Update Pattern Tracker using *recordOverResult(...)* to maintain outcome statistics.

**Time complexity:**

* String comparison: O(1)
* *randomBetween(...)*: O(1)
* *generateDiceSum(targetSum)*: Worst-case O(K), with K = number of dice combinations (bounded)
* *recordOverResult(...)*: O(1)
* Total: O(1)

## **4.4 stats**

### **a. StatisticsManager.java**

The *StatisticsManager* class is responsible for collecting and reporting game simulation statistics. It tracks total rounds played, player win/loss counts, frequency of each possible dice outcome (sum of three dice: 3–18). This information is essential for validating game fairness, assessing bias logic, and analyzing strategy performance.

**Core Data Structures**

|  |
| --- |
| private int totalRounds = 0;  private int playerWins = 0;  private int playerLosses = 0;  private final Map<Integer, Integer> diceOutcomeFrequency = new HashMap<>(); |

* **private int totalRounds;**: Tracks the total number of rounds played. Used for calculating the win rate and overall simulation scope.
* **private int playerWins;**: Stores the count of rounds the player has won. Used to compute win rate and assess strategy effectiveness.
* **private int playerLosses;**: Stores the count of rounds the player has lost. Complements win count for fair outcome evaluation.
* **private final Map<Integer, Integer> diceOutcomeFrequency;**: A HashMap that maps each possible dice total (3–18) to the number of times it has occurred. Used to analyze distribution fairness and potential bias trends in dice outcomes.

**Core Functionalities**

1. **Core Method**

|  |
| --- |
| public void recordOutcome(int diceTotal, boolean playerWon) {  totalRounds++;  if (playerWon) {  playerWins++;  } else {  playerLosses++;  }  diceOutcomeFrequency.put(diceTotal, diceOutcomeFrequency.getOrDefault(diceTotal, 0) + 1); } |

* Updates counters in constant time: O(1)
* Frequency map update uses *getOrDefault()* and *put()* - both average O(1)
* Efficient for real-time simulations (even with large datasets)

1. **Summary Method**

|  |
| --- |
| public void printStatistics() {  System.*out*.println("\n=== Game Statistics ===");  System.*out*.println("Total rounds played: " + totalRounds);  System.*out*.println("Player wins: " + playerWins);  System.*out*.println("Player losses: " + playerLosses);  double winRate = totalRounds == 0 ? 0.0 : (double) playerWins / totalRounds \* 100;  System.*out*.printf("Win rate: %.2f%%\n", winRate);  System.*out*.println("\nDice Outcome Frequency:");  for (int total = 3; total <= 18; total++) {  int count = diceOutcomeFrequency.getOrDefault(total, 0);  System.*out*.printf("Sum %2d: %d\n", total, count);  }  System.*out*.println("========================\n"); } |

**Algorithm:**

* Prints all stats including win rate
* Loops through the fixed range of valid dice totals (3 to 18)
* Fetches frequencies from *HashMap* (O(1) per key)

**Time complexity:**

* Fixed loop of 16 iterations → O(1)

## **4.5 engine**

### **a. GameEngine.java**

GameEngine serves as the main logic controller for each game round. It coordinates between the Player, House, and BiasManager to simulate the outcome of a betting round. It encapsulates the following responsibilities:

* Roll dice (with bias logic)
* Evaluate win/loss conditions
* Update player and house balances
* Track game state after each round

**Core Data Structures**

* **private final House house;**: Stores a reference to the shared House object that manages the casino’s balance. Ensures centralized fund tracking.
* **private final BiasManager biasManager;**: Integrates advanced bias control, pattern detection, and player-counter strategies for dynamic dice manipulation.
* **private int[] diceValues = new int[3];**: Stores the outcome of the latest dice roll for reference and display. Constant size, fixed memory usage.
* **private boolean playerWins;**: Boolean flag representing whether the player won the last round. Used by the UI and external systems.
* **private String resultMessage;**: Holds a textual description of the result (e.g., "You win! Dice total: 14"). Returned for user display.

**Core Algorithmic Components**

|  |
| --- |
| public String playRound(Player player) {  if (!player.hasEnoughBalance()) {  return "Insufficient balance to place the bet.";  }  String playerBet = player.getBetChoice();  int diceTotal = biasManager.rollBiasedDice(playerBet, player.getInitialBalance(), player.getBalance(), player.getBetAmount());  diceValues = biasManager.getDiceValues();  playerWins = checkWin(diceTotal, playerBet);  if (playerWins) {  player.winBet();  house.loseMoney(player.getBetAmount());  } else {  player.loseBet();  house.winMoney(player.getBetAmount());  }  biasManager.updateStreak(playerWins);  biasManager.updateLastRoundResult(playerWins);  return resultMessage = playerWins ? "You win! Dice total: " + diceTotal : "You lose. Dice total: " + diceTotal; } private boolean checkWin(int sum, String choice) {  boolean isOverChoice = choice.equalsIgnoreCase("over");  boolean isActualOver = sum > 10; // 11-18  return (isOverChoice && isActualOver) || (!isOverChoice && !isActualOver); } |

**Algorithm:**

1. **Balance Validation**

|  |
| --- |
| if (!player.hasEnoughBalance()) {  return "Insufficient balance to place the bet.";  } |

* Purpose: Prevent execution if the player cannot cover the bet.
* Complexity: O(1) - Constant-time check using getter logic.

1. **Bet Choice Extraction**

|  |
| --- |
| String playerBet = player.getBetChoice(); |

* Purpose: Capture the player's current betting direction ("over" or "under").
* Complexity: O(1) - Simple field access.

1. **Biased Dice Roll**

|  |
| --- |
| int diceTotal = biasManager.rollBiasedDice(...);  diceValues = biasManager.getDiceValues(); |

* Purpose: Generate a dice roll using the *BiasManager*, which applies conditional bias.
* Algorithm: Internally invokes a strategy that adapts based on:
* Player profit margin
* Betting patterns
* Streaks
* Anti-pattern defense
* Complexity: O(1) externally, though bias logic includes sliding window checks and probabilistic control.

1. **Win Condition Check**

|  |
| --- |
| playerWins = checkWin(diceTotal, playerBet); |

Use:

|  |
| --- |
| private boolean checkWin(int sum, String choice) {  boolean isOverChoice = choice.equalsIgnoreCase("over");  boolean isActualOver = sum > 10;  return (isOverChoice && isActualOver) || (!isOverChoice && !isActualOver); } |

* Purpose: Determine game outcome based on sum logic.
* Complexity: O(1) - Simple conditionals.

1. **Payout Logic**

|  |
| --- |
| if (playerWins) {  player.winBet();  house.loseMoney(player.getBetAmount());  } else {  player.loseBet();  house.winMoney(player.getBetAmount());  } |

* Purpose: Transfer money between player and house based on result.
* Complexity: O(1) - Involves arithmetic operations and variable updates.

1. **Bias Feedback Update**

|  |
| --- |
| biasManager.updateStreak(playerWins);  biasManager.updateLastRoundResult(playerWins); |

* Purpose: Provide feedback to the *BiasManager* for adaptive behavior in future rounds.
* Complexity: O(1) for each - capped sliding windows and counters.

1. **Result Output**

|  |
| --- |
| return resultMessage = playerWins  ? "You win! Dice total: " + diceTotal  : "You lose. Dice total: " + diceTotal; |

* Purpose: Generate a user-facing summary of the round.
* Complexity: O(1) – Simple conditional formatting.

**Time complexity:**

All operations in playRound() are constant time → Overall: O(1).

## **4.6 resources**

The resources package contains external assets used in the game's visual interface. In this project, it includes Dice.png, an image representing dice faces. These graphical elements enhance user experience by providing intuitive and visually appealing feedback during gameplay, especially when displaying the result of each dice roll.

A white and blue square with black square

AI-generated content may be incorrect.

*Dice 1*

A white square with black squares

AI-generated content may be incorrect.

*Dice 2*



*Dice 3*

A white and blue square with black squares

AI-generated content may be incorrect.

*Dice 4*

A white and black dice

AI-generated content may be incorrect.

*Dice 6*

A white and blue dice

AI-generated content may be incorrect.

*Dice 6*

## **4.7 panel**

### **a. HomePanel.java**

HomePanel is the graphical entry point of the application, designed to collect user input including name and initial balance. It employs a structured layout using BorderLayout, BoxLayout, and GridLayout for clean alignment and separation of concerns.

**Core Design:**

* **Input Fields:**

|  |
| --- |
| nameField = new JTextField(15);  balanceField = new JTextField(15); |

* JTextField nameField for the player's name.
* JTextField balanceField for the initial balance.

|  |
| --- |
| if (name.isEmpty()) {  JOptionPane.showMessageDialog(this, "Please enter your name!", "Error", JOptionPane.ERROR\_MESSAGE);  } else if (balanceText.isEmpty()) {  JOptionPane.showMessageDialog(this, "Please enter a starting balance!", "Error", JOptionPane.ERROR\_MESSAGE);  } else {  try {  double balance = Double.parseDouble(balanceText);  if (balance <= 0) {  JOptionPane.showMessageDialog(this, "Balance must be greater than 0!", "Error", JOptionPane.ERROR\_MESSAGE);  return;  } |

* Validations ensure both fields are non-empty and the balance is a positive numeric value.
* **Event Handling:**

|  |
| --- |
| startButton.addMouseListener(new MouseAdapter() {  @Override  public void mouseEntered(MouseEvent e) {  startButton.setBackground(new Color(70, 170, 70)); // Hover  }  @Override  public void mouseExited(MouseEvent e) {  startButton.setBackground(new Color(50, 150, 50)); // Reset  }  }); |

* A JButton with mouse listeners adds hover effects.

|  |
| --- |
| startButton.addActionListener(e -> {  String name = nameField.getText().trim();  String balanceText = balanceField.getText().trim();  ...  mainFrame.setPlayerName(name);  mainFrame.setPlayerBalance(balance);  mainFrame.setGamePanel(new GamePanel(name, balance, mainFrame));  mainFrame.showPanel("gamePanel");  }); |

* The action listener checks inputs and initializes the GamePanel with valid data.
* **Layout Composition:**

|  |
| --- |
| titlePanel.setLayout(new BoxLayout(titlePanel, BoxLayout.Y\_AXIS));  titleLabel.setFont(new Font("Impact", Font.BOLD, 54)); |

* Top: Game title and dice image using BoxLayout for vertical alignment.

|  |
| --- |
| JPanel inputRowPanel = new JPanel(new GridLayout(1, 2, 20, 0)); |

* Center: A GridLayout with two columns, collecting name and balance input side-by-side.

|  |
| --- |
| centerPanel.add(startButton); |

* Bottom: A centered start button aligned using BoxLayout.

**Visual Enhancements:**

|  |
| --- |
| @Override  protected void paintComponent(Graphics g) {  super.paintComponent(g);  Graphics2D g2d = (Graphics2D) g;  Color color1 = new Color(27, 94, 149);  Color color2 = new Color(10, 50, 100);  GradientPaint gp = new GradientPaint(0, 0, color1, 0, getHeight(), color2);  g2d.setPaint(gp);  g2d.fillRect(0, 0, getWidth(), getHeight());  } |

* Gradient background is rendered by overriding *paintComponent()*, enhancing user experience with a polished interface.

**Algorithmic Complexity:**

* Input validation and layout updates occur in constant time, i.e., **O(1)**.
* No iterative computation or performance bottlenecks exist in this static UI component.

**Purpose and Role:**

* Acts as the gateway to the game logic by collecting initial parameters from the user.
* Bridges user input to *MainFrame* control logic, triggering a transition to the main gameplay screen.

### **b. GamePanel.java**

*GamePanel* serves as the main gameplay interface where users place bets, select their prediction (under/over), and view live dice animations and results. It tightly integrates with the game logic and displays current player state such as balance, bet amount, and win streak.

**Core Design:**

**Input Field:**

|  |
| --- |
| betField = new JTextField(10); |

* *JTextField betField*: For entering the desired bet amount.
* Integrated validation ensures numeric input and sufficient balance.

**Validation:**

|  |
| --- |
| String betText = betField.getText().trim();  double bet = Double.parseDouble(betText);  if (bet <= 0 || bet > player.getBalance()) {  JOptionPane.showMessageDialog(this, "Invalid bet amount!");  } |

* Ensures the entered bet is valid (non-negative and within balance).

**Bet Amount Buttons:**

Quick-select betting amounts update the bet field instantly:

|  |
| --- |
| five.addActionListener(e -> betField.setText("5"));  ten.addActionListener(e -> betField.setText("10"));  twenty.addActionListener(e -> betField.setText("20"));  fifty.addActionListener(e -> betField.setText("50"));  hundred.addActionListener(e -> betField.setText("100"));  allIn.addActionListener(e -> betField.setText(String.valueOf(player.getBalance()))); |

* Improves user speed and convenience in selecting common bets.

**Prediction Buttons ("Under"/"Over"):**

Initiates game round with the user's choice:

|  |
| --- |
| underButton.addActionListener(e -> playRound("under"));  overButton.addActionListener(e -> playRound("over")); |

* Validates bet, runs animation, evaluates win/loss, updates dice visuals and labels.

**Dice Animation and Outcome Display:**

Controlled animation loop using Timer:

|  |
| --- |
| Timer animationTimer = new Timer(100, null);  animationTimer.addActionListener(e -> {  updateDiceIcons(random.nextInt(6) + 1, random.nextInt(6) + 1, random.nextInt(6) + 1);  ...  if (count[0] >= 10) {  animationTimer.stop();  ...  JOptionPane.showMessageDialog(this, result);  }  }); |

* Simulates rolling dice over 10 frames before revealing result.
* Smooth animation with negligible performance cost (constant-time per frame).

**Exit Button:**

Graceful return to the main menu:

|  |
| --- |
| exitButton.addActionListener(e -> {  int confirm = JOptionPane.showConfirmDialog(this, "Exit to main menu?", "Confirm", JOptionPane.YES\_NO\_OPTION);  if (confirm == JOptionPane.YES\_OPTION) {  mainFrame.showPanel("homePanel");  }  }); |

* Adds session control and safety prompt before navigation.

**Hover and Styling:**

|  |
| --- |
| button.addMouseListener(new MouseAdapter() {  public void mouseEntered(MouseEvent e) {  button.setBackground(new Color(70, 170, 70));  }  public void mouseExited(MouseEvent e) {  button.setBackground(new Color(50, 150, 50));  }  }); |

* Buttons (like under/over and amount selectors) respond visually on hover for better UX.

**Betting and Game Execution:**

|  |
| --- |
| underButton.addActionListener(e -> playRound("under"));  overButton.addActionListener(e -> playRound("over")); |

* Triggers the round based on user choice.

**Layout Composition:**

**Top:**

|  |
| --- |
| JLabel welcomeLabel = new JLabel("Place a Bet, " + playerName + "!");  topPanel.add(welcomeLabel, BorderLayout.CENTER); |

* Displays player greeting and integrates an exit button.

**Center:**

|  |
| --- |
| centerPanel.setLayout(new BoxLayout(centerPanel, BoxLayout.Y\_AXIS)); |

* Dice visual feedback (three dice labels).
* Displays result bar and win streak tracking.

**Bottom:**

|  |
| --- |
| inputPanel.add(betField);  choicePanel.add(underButton);  choicePanel.add(overButton); |

* Bet input, quick bet buttons (e.g., “5$”, “All In”), and choice buttons (under/over).

**Visual Enhancements:**

|  |
| --- |
| @Override  protected void paintComponent(Graphics g) {  GradientPaint gp = new GradientPaint(0, 0, color1, 0, getHeight(), color2);  g2d.setPaint(gp);  g2d.fillRect(0, 0, getWidth(), getHeight());  } |

* Gradient background gives a smooth polished look, improving immersion.

**Algorithmic Complexity:**

* O(1) layout updates and component access.
* O(1) bet validation and state change.
* Uses a *Timer* and short animation loop for dice simulation:

|  |
| --- |
| Timer animationTimer = new Timer(100, null); |

* Controlled animation length (10 iterations) avoids unnecessary performance cost.

**Purpose and Role:**

* Acts as the primary game logic and interaction surface.
* Enables user input, ties into *GameEngine* for outcome calculation.
* Displays result via *ResultBar*, dice animations, and updated labels.
* Interfaces tightly with *StatisticsManager*, *Player*, *House*, and *MainFrame*.

### **c. ResultBar.java**

The ResultBar is a minimalist visual component that graphically displays the outcomes of recent game rounds in a rolling history bar. It enhances user engagement by visually indicating patterns (e.g., streaks of "Over" or "Under" results).

**Core Design**

**Result Storage:**

|  |
| --- |
| private final List<GameResult> lastResults = new ArrayList<>(); |

* Stores the most recent 15 game results.
* Acts as a sliding window data structure.

|  |
| --- |
| if (lastResults.size() >= 15) {  lastResults.remove(0); // Remove oldest entry  }  lastResults.add(new GameResult(diceTotal, isOver)); |

* Manual queue behavior simulates a fixed-size circular buffer (FIFO).
* Time complexity:
* *addResult*: O(1) amortized (single removal + append).
* Memory bounded by 15 elements → O(1) space.

**GameResult Data Structure:**

|  |
| --- |
| static class GameResult {  int diceTotal;  boolean isOver;  } |

* Encapsulates metadata of each round: total dice value and category ("Over"/"Under").
* Lightweight DTO (Data Transfer Object), ideal for short-term visual caching.

**paintComponent(Graphics g)**

|  |
| --- |
| for (int i = 0; i < lastResults.size(); i++) {  ...  g2d.fillOval(x, y, circleDiameter, circleDiameter);  ...  g2d.drawString(text, ...);  } |

* Draws each result as a circle with:
* White fill for "Over" results.
* Black fill for "Under" results.
* Contrasting text for dice sum inside the circle.
* Centers the entire bar horizontally using calculated startX.

**Color and Accessibility:**

* Contrast-aware design: Text and fill alternate to ensure readability:
* Over → white circle, black text.
* Under → black circle, white text.

**Font and Alignment:**

|  |
| --- |
| FontMetrics fm = g2d.getFontMetrics();  int textWidth = fm.stringWidth(text);  int textHeight = fm.getAscent(); |

* Text centered vertically and horizontally inside each circle.
* Font: Bold Arial for visibility, constant 16pt size.

**Algorithmic Complexity**

* Rendering loop: O(N), where N = number of stored results (max 15).
* Memory: Bounded to 15 GameResult objects → O(1) space.
* Repaint trigger: Only invoked on new result addition via repaint().

**Purpose and Role**

* Enhances player experience with clear visual feedback.
* Highlights short-term trends, such as long runs of “over” results.
* Aesthetic and functional, it fits seamlessly within GamePanel.

### **d. MainFrame.java**

*MainFrame* serves as the central controller and root window of the GUI-based dice game application. It orchestrates screen transitions between panels, manages core user data, and initializes the application window.

**Core Design**

**User Data Storage:**

|  |
| --- |
| private String playerName;  private double playerBalance; |

* Holds stateful information about the current player.
* Shared between *HomePanel* (input) and *GamePanel* (game logic).
* Accessors:
* *getPlayerName(), getPlayerBalance()*
* *setPlayerName(...), setPlayerBalance(...)*

**Panel Switching with CardLayout:**

|  |
| --- |
| private final CardLayout cardLayout;  private final JPanel cardPanel; |

* CardLayout is used to toggle between different screens (homePanel, gamePanel) in the same frame.
* The layout manager ensures only one panel is visible at a time.
* This design enables efficient screen transitions.

|  |
| --- |
| public void showPanel(String name) {  cardLayout.show(cardPanel, name);  } |

* Simplifies switching between views like "homePanel" and "gamePanel" using identifiers.

**Screen Construction & Initialization**

|  |
| --- |
| public MainFrame() {  cardLayout = new CardLayout();  cardPanel = new JPanel(cardLayout);  setTitle("Under or Over Dice Game - DSA Project");  setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  setSize(720, 480);  HomePanel homePanel = new HomePanel(this);  cardPanel.add(homePanel, "homePanel");  add(cardPanel);  setLocationRelativeTo(null);  } |

* Initializes the GUI with a fixed window size (720x480).
* Adds *HomePanel* as the default screen.
* Centers the frame on screen using *setLocationRelativeTo(null)*.

**Dynamic Panel Injection**

|  |
| --- |
| public void setGamePanel(GamePanel gamePanel) {  cardPanel.add(gamePanel, "gamePanel");  cardLayout.show(cardPanel, "gamePanel");  } |

* Dynamically loads the game interface after user input is validated in *HomePanel*.
* Ensures *GamePanel* is lazily initialized only when necessary.

**Application Entry Point**

|  |
| --- |
| public static void main(String[] args) {  SwingUtilities.invokeLater(() -> {  new MainFrame().setVisible(true);  });  } |

* Launches the UI safely on the Event Dispatch Thread (EDT), adhering to Swing’s threading model.

**Algorithmic Complexity**

* UI operations (add, show, set) are constant time: O(1).
* No data-intensive operations or performance bottlenecks.

**Purpose and Role**

* Acts as the root container for all UI panels.
* Maintains global player state (name, balance).
* Coordinates transitions between input screen (*HomePanel*) and game logic interface (*GamePanel*).
* Serves as the main entry point for the dice game application.

## **4.8 Strategy Explain**

### **a. Behavior-Triggered Bias**

The system’s bias strategy begins with monitoring player behavior. It detects repetitive betting patterns, such as choosing “over” for multiple rounds, and treats these as exploitable habits. If detected, there’s a 70% chance bias is activated to make the player lose. Martingale behavior - increasing bets after losses - is also tracked and countered, as the system recognizes the player is trying to recover losses aggressively. Moreover, when the player’s balance doubles, bias is guaranteed to activate, reversing their profit trend. This system uses a dynamic trigger mechanism that responds to predictability, profit level, and emotional betting to apply pressure precisely when the player is most vulnerable to loss.

### **b. Risk-Weighted Dice Control**

Bias is not applied uniformly; it adapts based on how risky the bet is. If a player bets more than 40% of their balance, the system almost always intervenes to cause a loss. This ensures maximum profit is extracted during the riskiest decisions. In contrast, small bets - under 10% of the balance - are often allowed to win naturally (with a 25% chance), especially when bias is disabled. This keeps the game believable and gives players emotional wins. The strategy is surgical: allow harmless wins to maintain engagement, and step in forcefully when players bet high, maximizing house advantage while keeping manipulation undetected.

A graph of a number of numbers

AI-generated content may be incorrect.

### **c. Hiding Manipulation**

To remain undetectable, the system limits how often it applies bias. After four consecutive bias-induced losses, bias is paused to restore fairness temporarily. This cooldown gives players hope and reduces the chance of them recognizing patterns. The system also monitors the last several outcomes, and if too many of them fall on the same side (e.g., “over” five times in a row), it disables bias to break the sequence. This ensures the results remain statistically varied, masking manipulation. The strategic intent is clear: maintain player trust through controlled randomness while continuing to guide outcomes in favor of the house behind the scenes.

### **d. Psychological Manipulation for Longevity**

Emotional conditioning is at the heart of the system. Players are allowed to win occasionally - especially during small bets or when bias is on cooldown - to build false confidence. These small, frequent wins create the illusion of fairness and encourage players to keep betting. However, large bets, repetitive strategies, or emotional plays after losing streaks are targeted with bias. This pattern mimics addictive reinforcement cycles, where players chase previous wins but encounter losses precisely when they risk more. The player feels like winning is possible if they just try again - but in reality, the system is engineered to ensure long-term defeat while keeping them emotionally invested.

## **4.9 Result**

**Simulator result:**

|  |
| --- |
| ===== SIMULATION STATS =====  === Game Statistics ===  Total rounds played: 1331  Player wins: 591  Player losses: 740  Win rate: 44.40%  Dice Outcome Frequency:  Sum 3: 38  Sum 4: 56  Sum 5: 66  Sum 6: 83  Sum 7: 108  Sum 8: 85  Sum 9: 135  Sum 10: 99  Sum 11: 115  Sum 12: 118  Sum 13: 126  Sum 14: 85  Sum 15: 65  Sum 16: 53  Sum 17: 52  Sum 18: 47  ========================  Final Player Balance: $0.00  Final House Balance: $2000.00 |

Win Rate: At 44.40%, the player's win rate is notably below the expected 50% in a fair system, indicating successful intervention by the BiasManager.

Player Balance: Reached $0.00, while the House ended with $2000.00, suggesting the house consistently profited over time.

The simulator effectively demonstrates how bias and pattern manipulation can skew odds against players, emphasizing the project's educational message about the exploitative nature of gambling systems.

**Counter Martingale strategy:**

When I set :

|  |
| --- |
| initial\_balance = 10000  base\_bet = 1  rounds = 100000  MAX\_BIAS\_LOSSES = 4  bias\_probability = 0.6  bias\_active\_probability = 0.4 |

A graph with blue lines

AI-generated content may be incorrect.

100 rounds

A graph with blue lines

AI-generated content may be incorrect.

1000 rounds

A graph with blue lines

AI-generated content may be incorrect.

10000 rounds

A graph with blue lines

AI-generated content may be incorrect.

100000 rounds

The simulation clearly demonstrates that the Martingale strategy fails under biased conditions. Despite starting with a large initial balance ($10,000), the player's balance eventually trends toward zero due to compounding losses and bias enforcement. Profitability under the Martingale strategy emerges only after an extensive number of rounds, and even then, the returns remain modest relative to the risk. For example, simulations show a gain of approximately $40 after 100 rounds, $440 after 1,000 rounds, and around $4,700 after 10,000 rounds. Although the trend appears upward, the strategy’s dependence on uninterrupted capital and its vulnerability to extended loss streaks make it unsustainable in the long term—particularly under biased or constrained conditions.

# **CHAPTER 5: FINAL APP GAME**

## **5.1 Source code (link github):**

Github: [DuNhan1930/Project\_DSA](https://github.com/DuNhan1930/Project_DSA)

## **5.2 Demo video:**

|  |
| --- |
| [Project\_DSA.mp4 - Google Drive](https://drive.google.com/file/d/16JRLqNc7qKDhIcDkR3p-K1YdiBeshJw9/view) |

## **5.3 Instruction:**

### **a. Main Menu**

A screenshot of a game

AI-generated content may be incorrect.

On the home screen, enter your player’s name and an initial balance to begin. Both fields are required to proceed.

### **b. Game Panel**

Once the game starts, input your desired bet amount or use quick bet buttons like 5$, 10$, 20$, etc.

Then, choose either:

* **Under (3–10)**
* **Over (11–18)**

Based on what you predict the total of the three dice will be.

A screenshot of a video game

AI-generated content may be incorrect.

### **c. Game Play**

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

If your guess is wrong, the message will show "You lose.", and your balance will decrease accordingly.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

In contrast, if your prediction is correct, a message will appear saying "You win!", and your balance will increase by the amount you bet.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

To exit, click the "Exit" button at the top-left corner, then confirm by selecting "Yes".

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

A screenshot of a game

AI-generated content may be incorrect.

If you lose all your money, a message will notify you.  
To try again, simply press "Exit" and return to the home screen to restart with a new balance.

# **CHAPTER 6: EXPERIENCE**

Developing this project offered valuable hands-on experience in applying data structures and algorithms within a real-world context. Implementing features like bias detection, streak tracking, and betting strategy simulations deepened my understanding of concepts such as sliding windows (using LinkedList) and time complexity analysis (e.g., O(1) operations for updating streaks). Integrating backend logic with a responsive Swing-based GUI further enhanced my practical skills in event-driven programming and user interface design. Moreover, simulating gambling mechanics provides insight into how statistical manipulation can influence perceived fairness and outcomes. Through testing and iteration, I improved not just my technical proficiency, but also my awareness of ethical software design—recognizing the psychological impacts of systems designed to influence user behavior.

*Gambling is not good.*

*This project is built for educational purposes only, simulating how real-world gambling systems may exploit players.*

*The only true winner in gambling is the one who never plays.*