Design Document
Group 13

Jyotsna Venkatesan: 22108825D
Tahmin Anower : 22107453D
Tanya Budhrani: 22097189D
Tanchhoma Limbu: 22040966D

INDEX

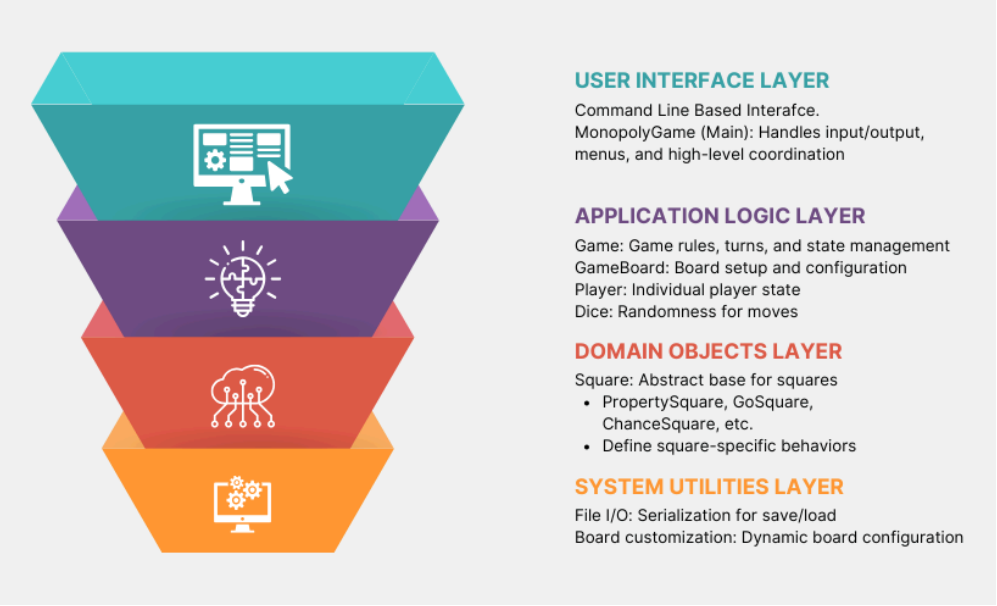
1. Architectural Pattern: Layered Architecture

The Monopoly game is designed using the **Layered Architecture** pattern. This pattern is widely used for its ability to separate concerns into distinct layers, each responsible for a specific functionality in the system. Each layer depends only on the layer immediately below it, ensuring modularity and flexibility.

1.1. Why Did We Choose Layered Architecture?

1. **Separation of Concerns**: Each layer handles distinct responsibilities, which makes the code easier to understand, debug, and maintain.
2. **Modularity**: Layers can be developed and tested independently without impacting other parts of the system.
3. **Scalability**: New features (e.g., new types of squares or a graphical interface) can be added by modifying or extending specific layers.
4. **Reusability**: Core game logic and domain objects can be reused in other variants of the Monopoly game.
5. **Ease of Maintenance**: Changes to the game logic (e.g., modifying rules for jail) or the user interface (e.g., switching to a GUI) require changes only in specific layers.

1.2 Diagram



1.3 Breakdown of the Layers of Monopoly Game Architecture

The architecture is divided into four main layers, each with specific responsibilities:

Layer 1: User Interface Layer

- **Components**: MonopolyGame (Main class)
- **Responsibilities**:
    - Provides the console-based interface for the player to interact with the game.
    - Displays menus and prompts for user input (e.g., rolling dice, choosing actions).
    - Handles high-level game orchestration, such as starting a new game, loading a game, or customizing the board.
- **Example**

```
Game game = new Game(playerNames);
playGame(game);
```

Layer 2: Application Logic Layer

- **Components**: Game, GameBoard, Player, Dice
- **Responsibilities**:
    - Manages the core functionality of the game:
        - Turns, rounds, dice rolls, player movements.
        - Game rules (e.g., paying rent, handling bankruptcy).
    - Maintains the overall state of the game, including the current round, active players, and board configuration.
    - Encapsulates game mechanics and ensures the proper sequence of actions during play.
- **Example**:
    - **Dice rolls and player movement**:

```
int roll1 = dice1.roll();
int roll2 = dice2.roll();
int totalRoll = roll1 + roll2;
player.setPosition(player.getPosition() + totalRoll);
board[newPosition].landOn(player, this);
```

- **Handling turns**:

```
public void takeTurn(Player player) {
    if (player.isInJail()) {
        handleJailTurn(player);
        return;
    }
    int roll1 = dice1.roll();
    int roll2 = dice2.roll();
    player.setPosition(player.getPosition() + roll1 +
roll2);
    board[player.getPosition()].landOn(player, this);
}
```

Layer 3: Domain Objects Layer

- **Components**: Square and its subclasses (PropertySquare, GoSquare, JailSquare, etc.)
- **Responsibilities**:
  - Represents game-specific entities like properties, chance squares, and tax squares.
  - Encapsulates the behavior of each type of square:
    - What happens when a player lands on or passes by the square.
    - Ownership and rent mechanics for properties.
    - Special effects of chance or tax squares.
- **Example**:
  - **PropertySquare behavior**:

```
public void landOn(Player player, Game game) {
    if (owner == null && player.getMoney() >= price) {
        player.reduceMoney(price);
        owner = player;
    } else if (owner != null && owner != player) {
        player.reduceMoney(rent);
        owner.addMoney(rent);
    }
}
```

- **ChanceSquare random effects**:

```
public void landOn(Player player, Game game) {
    if (random.nextBoolean()) {
        player.addMoney(gain);
    } else {
        player.reduceMoney(loss);
    }
}
```

Layer 4: System Utilities Layer

- **Components**: File I/O operations for saving and loading games (Serializable utilities).
- **Responsibilities**:
  - Handles persistence of game and board state for resuming play.
  - Provides mechanisms for saving and loading game data.
- **Example**:
  - **Saving game state**:

```
public void saveGame(String fileName) throws IOException {
    try (ObjectOutputStream out = new
ObjectOutputStream(new FileOutputStream(fileName))) {
        out.writeObject(this);
    }
}
```

  - **Loading game state**:

```
public static Game loadGame(String fileName) throws
IOException, ClassNotFoundException {
    try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream(fileName))) {
        return (Game) in.readObject();
    }
}
```

1.4 Instantiation of Components

- **User Interface Layer**:
    - The MonopolyGame class initializes and orchestrates the game. For example:
        - It creates a new Game instance with player names.
        - It delegates gameplay actions to the Game class.
        - It handles menu navigation and user prompts.
- **Application Logic Layer**:
    - The Game class manages the sequence of play, invokes dice rolls, moves players, and processes square interactions.
    - The GameBoard class initializes and stores the board configuration.
    - The Player and Dice classes handle individual player state and randomness in gameplay.
- **Domain Objects Layer**:
    - The Square class hierarchy defines all square types, encapsulating their specific behaviors.
    - The PropertySquare class handles property mechanics like buying, renting, and ownership.
- **System Utilities Layer**:
    - Serialization utilities (saveGame, loadGame) store and retrieve game state from files.
    - Board customization options allow dynamic configuration and saving of new board layouts.

1.5 Layered Architecture Conclusion

The **Layered Architecture** is ideal for the Monopoly game due to its emphasis on modularity, separation of concerns, and scalability. Each layer has a clear role, allowing independent updates and extensions. This architecture ensures that the Monopoly game is maintainable, extensible, and ready for enhancements like a graphical UI or advanced game rules.

★ Programming Technique

The provided code adopts **Object-Oriented Programming (OOP)** principles. Here are the features in the code that reflect OOP techniques:

1. Encapsulation

- Classes like Square, Player, Game, Dice, and GameBoard encapsulate related data (fields) and methods. For example:
    - Player has fields like name, money, and position with corresponding methods (addMoney, reduceMoney, etc.) to interact with those fields.
    - Square has protected data (position, name) with methods like landOn to ensure interaction is controlled.

2. Inheritance

- The Square class serves as a base (abstract) class, and specific types of squares (like GoSquare, PropertySquare, ChanceSquare, etc.) extend it, inheriting its fields and methods.
- This allows for **code reuse** and modular design where different types of squares implement their own behavior for the landOn method.

3. Polymorphism

- The code uses **method overriding** in subclasses. For example:
    - The landOn method is overridden in subclasses like PropertySquare and ChanceSquare to implement behavior specific to those squares.
    - The passBy method in GoSquare adds a specific functionality while being optional for other square types.
- This enables **dynamic behavior** where the actual method called depends on the type of object being referred to, not the type of reference.

4. Abstraction

- Abstract methods like landOn in the Square class provide a **blueprint** for child classes, ensuring that they implement their unique behavior for a player landing on the square.

2. Code Components

★2.1 Abstract Class: Square

Implements:Serializable

Fields:

1. **position: (private, final, int)**
    - Represents the position of the square on the game board.

2. **name: (private, final, String)**
    - Represents the name of the square.

Constructor:

1. **Square(int position, String name)**
    - **Parameters**:
        - **position (int):** The position of the square.
        - **name (String):** The name of the square.
    - **Description**:
        - Initializes the position and name fields to their provided values.

Methods:

1. **getPosition()**
    - **Access Modifier**: public

- **Return Type**: int
- **Description**: Returns the position of the square.

2. getName()**.Access Modifier**: public
    - **Return Type**: String
    - **Description**: Returns the name of the square.

3. landOn(Player player, Game game)**.**
    - **Access Modifier**: abstract
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**: Abstract method to define the behavior when a player lands on the square.

4. passBy(Player player, Game game)**.**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player passing by the square.
        - game (Game): The current game state.
    - **Description**: Provides a default implementation for actions when a player passes by the square. By default, it does nothing.

★ 2.2 Class: PropertySquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
    - Ensures compatibility during deserialization.

2. **price**: (private, final, int)
    - Represents the cost to purchase the property.

3. **rent**: (private, final, int)
    - Represents the rent amount to be paid by others who land on this square.

4. **owner**: (private, Player)
    - The current owner of the property (null if unowned).

Constructor:

1. **PropertySquare(int position, String name, int price, int rent)**
    - **Parameters**:
        - position (int): The position of the property on the board.
        - name (String): The name of the property.
        - price (int): The cost to purchase the property.
        - rent (int): The rent charged to other players.
    - **Description**:
        - Initializes the position, name, price, and rent fields by calling the Square constructor and setting the respective values.

Methods:

1. **Override Method: landOn(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.

        - game (Game): The current game state.
    - **Description**:
        - Handles the behavior when a player lands on the property:
            - If the property is unowned and the player has enough money:
                - Prompts the player to decide whether to buy the property.
                - Deducts the purchase price if the player agrees and sets the player as the owner.
            - If the property is owned by another player:
                - Deducts rent from the current player and credits it to the owner.
            - If the property is owned by the same player:
                - No action is taken.
    - **Exceptions**:
        - None, but invalid inputs for purchase prompt are handled within the method.

2. **Getter: getOwner()**
    - **Access Modifier**: public
    - **Return Type**: Player
    - **Description**: Returns the current owner of the property (or null if unowned).

3. **Setter: setOwner(Player owner)**
    - **Access Modifier**: public
    - **Return Type**: void
    - **Parameters**:
        - owner (Player): The new owner of the property.

★ 2.3Class: GoSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
    - Ensures compatibility during deserialization.

2. **SALARY**: (private, static, final, int)
    - Represents the fixed salary amount awarded to players when they pass or land on this square. Value is set to 1500.

Constructor:

1. **GoSquare()**
    - **Parameters**: None
    - **Description**:
        - Calls the superclass (Square) constructor to initialize the position to 1 and name to "Go".

Methods:

1. **Override Method: landOn(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**:
        - Adds the SALARY amount to the player's balance.
        - Prints a message indicating that the player received the salary for landing on the square.

2. **Override Method: passBy(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void

- **Parameters**:
  - player (Player): The player passing by the square.
  - game (Game): The current game state.
- **Description**:
  - Adds the SALARY amount to the player's balance if they pass the "Go" square but are not already on it.
  - Prints a message indicating that the player received the salary for passing the square.

★ 2.4 Class: ChanceSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
   - Ensures compatibility during deserialization.

2. **random**: (private, final, Random)
   - Used to generate random outcomes when a player lands on this square.

Constructor:

1. **ChanceSquare(int position)**
   - **Parameters**:
     - position (int): The position of the square on the board.
   - **Description**:
     - Calls the superclass (Square) constructor to initialize the position and sets the name of the square to "Chance".

Methods:

1. **Override Method: landOn(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**:
        - Uses the random field to determine a 50% chance for either:
            - **Gain Scenario**:
                - Calculates a random gain between $10 and $200 (multiples of $10).
                - Adds the gain amount to the player's balance.
                - Prints a message indicating the gain.
            - **Loss Scenario**:
                - Calculates a random loss between $10 and $300 (multiples of $10).
                - Deducts the loss amount from the player's balance.
                - Prints a message indicating the loss.

★ 2.5 Class: IncomeTaxSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
    - Ensures compatibility during deserialization.

Constructor:

1. **IncomeTaxSquare(int position)**
    - **Parameters**:
        - position (int): The position of the square on the board.
    - **Description**:
        - Calls the superclass (Square) constructor to initialize the position and sets the name of the square to "Income Tax".

Methods:

1. **Override Method: landOn(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**:
        - Prints a message indicating the player landed on the "Income Tax" square.
        - Calculates a tax of **10% of the player's total money**, rounded down to the nearest multiple of 10.
        - Deducts the calculated tax amount from the player's balance using player.reduceMoney(tax).
        - Prints a message specifying the amount of tax paid.

★ 2.6 Class: JailSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
    - Ensures compatibility during deserialization.

Constructor:

1. **JailSquare()**
    - **Parameters**: None
    - **Description**:
        - Calls the superclass (Square) constructor to initialize the position to 11 and sets the name of the square to "In Jail/Just Visiting".

Methods:

1. **Override Method: landOn(Player player, Game game)**
    - **Access Modifier**: void
    - **Return Type**: void
    - **Parameters**:
        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**:
        - Checks if the player is not in jail using player.isInJail().
            - If the player is not in jail, prints a message indicating the player is "just visiting jail."
            - If the player is in jail, no action is taken (as jail mechanics are handled elsewhere in the game logic).

★ 2.7 Class: GoToJailSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
    - Ensures compatibility during deserialization.

Constructor:

1. **GoToJailSquare(int position)**
    - **Parameters**:
        - position (int): The position of the square on the board.
    - **Description**:
        - Calls the superclass (Square) constructor to initialize the position and sets the name of the square to "Go to Jail".
    - **Exceptions**: None

Methods:

1. **Override Method: landOn(Player player, Game game)**

- **Access Modifier**: void
- **Return Type**: void
- **Parameters**:
  - player (Player): The player landing on the square.
  - game (Game): The current game state.
- **Description**:
  - Sends the player to jail by calling player.goToJail().
  - Updates the player's position to the jail square (position 11) using player.setPosition(11).
  - Prints a message indicating the player has been sent to jail.

★ 2.8 Class: FreeParkingSquare

Extends: Square
Implements: Serializable

Fields:

1. **serialVersionUID**: (private, static, final, long)
   - Ensures compatibility during deserialization.

Constructor:

1. **FreeParkingSquare(int position)**
   - **Parameters**:
     - position (int): The position of the square on the board.
   - **Description**:
     - Calls the superclass (Square) constructor to initialize the position and sets the name of the square to "Free Parking".

Methods:

1. **Override Method: landOn(Player player, Game game)**
   - **Access Modifier**: void
   - **Return Type**: void
   - **Parameters**:

        - player (Player): The player landing on the square.
        - game (Game): The current game state.
    - **Description**:
        - Prints a message indicating that the player has landed on "Free Parking".
        - Specifies that no actions occur as a result of landing on this square.

★ 2.9 Class: Player

Implements: Serializable

Fields:

1. **name**: (private, final, String)
    - The name of the player.

2. **money**: (private, int)
    - The player's current balance, initialized to $1500.

3. **position**: (private, int)
    - The player's current position on the board, initialized to 1.

4. **inJail**: (private, boolean)
    - Indicates whether the player is currently in jail.

5. **turnsInJail**: (private, int)
    - Tracks the number of turns the player has spent in jail.

Constructor:

1. **Player(String name)**
    - **Parameters**:
        - name (String): The player's name.
    - **Description**:
        - Initializes the player's name, starting balance (1500), position (1), and jail status (false).
    - **Exceptions**: None

Methods:

1. **Getter: getName()**
    - **Access Modifier**: public
    - **Return Type**: String
    - **Description**: Returns the player's name.

2. **Getter: getMoney()**
    - **Access Modifier**: public
    - **Return Type**: int
    - **Description**: Returns the player's current money balance.

3. **Getter: getPosition()**
    - **Access Modifier**: public
    - **Return Type**: int
    - **Description**: Returns the player's current position on the board.

4. **Getter: isInJail()**
    - **Access Modifier**: public
    - **Return Type**: boolean
    - **Description**: Returns true if the player is in jail, otherwise false.

Actions and Game State Methods:

5. **addMoney(int amount)**

- **Access Modifier**: public
- **Return Type**: void
- **Parameters**:
  - amount (int): The amount to add to the player's balance.
- **Description**: Increases the player's money by the specified amount.

6. **reduceMoney(int amount)**
   - **Access Modifier**: public
   - **Return Type**: void
   - **Parameters**:
     - amount (int): The amount to deduct from the player's balance.
   - **Description**:
     - Reduces the player's money by the specified amount.
     - Prints a message if the player's balance goes negative, indicating bankruptcy.

7. **setPosition(int position)**
   - **Access Modifier**: public
   - **Return Type**: void
   - **Parameters**:
     - position (int): The new position for the player on the board.
   - **Description**: Updates the player's current position.

8. **goToJail()**
   - **Access Modifier**: public
   - **Return Type**: void
   - **Description**:
     - Sets the player's jail status to true.
     - Resets the turnsInJail counter to 0.

Jail-Related Methods:

9. **tryToGetOutOfJail(int dice1, int dice2)**
    - **Access Modifier**: public
    - **Return Type**: boolean
    - **Parameters**:
        - dice1 (int): The first dice roll.
        - dice2 (int): The second dice roll.
    - **Description**:
        - Increments the turnsInJail counter.
        - If the dice rolls are doubles or the player has spent 3 turns in jail:
            - Sets the player's jail status to false.
            - Deducts $150 if released after 3 turns.
            - Resets the turnsInJail counter and returns true.
        - Otherwise, returns false.

10. **payJailFine()**
    - **Access Modifier**: public
    - **Return Type**: void
    - **Description**:
        - Checks if the player has sufficient money to pay a $150 fine.
        - Deducts the fine, sets the jail status to false, and resets turnsInJail.
        - Prints a message indicating the player paid the fine.

Status Method:

11. **getStatus()**
    - **Access Modifier**: public
    - **Return Type**: String
    - **Description**:
        - Returns a formatted string showing:
            - The player's name.
            - Current money balance.
            - Current position.
            - Jail status (if applicable).

★ 2.10 Class: Dice

Implements: Serializable

Fields:

1. **random**: (private, final, Random)
    - An instance of Random used to generate random dice rolls.

Constructor:

- **Default Constructor**: Implicit (not explicitly defined)
    - **Description**:
        - Uses the default constructor provided by Java.
        - No parameters or custom initialization required.

Methods:

1. **roll()**
    - **Access Modifier**: public
    - **Return Type**: int
    - **Description**:
        - Simulates rolling a 4-sided die.
        - Generates a random integer between 1 and 4 (inclusive).
        - The range is achieved by generating a number between 0 and 3 (using random.nextInt(4)) and adding 1 to the result.
    - **Parameters**: None
    - **Exceptions**: None

★ 2.11 Class: GameBoard

Implements: Serializable

Fields:

1. **squares**: (private, Square[])
    - An array of Square objects representing the board.
    - The array uses 1-based indexing (positions 1 to 20).

Constructor:

1. **GameBoard()**
    - **Parameters**: None
    - **Description**:
        - Initializes the squares array with 21 elements (1-based indexing).
        - Calls the initializeDefaultBoard method to populate the board with default squares.

Methods:

Utility Methods:

1. **initializeDefaultBoard()**
    - **Access Modifier**: private
    - **Return Type**: void
    - **Description**:
        - Placeholder for logic to initialize the board with default squares.
        - Currently has no implementation.

2. **setSquare(int position, Square square)**
    - **Access Modifier**: public
    - **Return Type**: void
    - **Parameters**:
        - position (int): The position on the board where the square should be placed.

        - square (Square): The square object to set at the specified position.
    - **Description**:
        - Validates the position to ensure it is between 1 and 20.
        - Sets the specified square at the given position in the squares array.
    - **Exceptions**:
        - Throws IllegalArgumentException if the position is out of range.

3. **getSquare(int position)**
    - **Access Modifier**: public
    - **Return Type**: Square
    - **Parameters**:
        - position (int): The position of the square to retrieve.
    - **Description**:
        - Returns the square at the specified position in the squares array.

Save/Load Methods:

4. **saveBoard(String fileName)**
    - **Access Modifier**: public
    - **Return Type**: void
    - **Parameters**:
        - fileName (String): The name of the file to save the board to.
    - **Description**:
        - Serializes the GameBoard object to the specified file.
    - **Exceptions**:
        - Throws IOException if an error occurs during file writing.

5. **loadBoard(String fileName)**
    - **Access Modifier**: public, static
    - **Return Type**: GameBoard
    - **Parameters**:
        - fileName (String): The name of the file to load the board from.
    - **Description**:

        - Deserializes the GameBoard object from the specified file.
    - **Exceptions**:
        - Throws IOException or ClassNotFoundException if an error occurs during file reading.

★ 2.12 Class: MonopolyGame

Fields:

1. **scanner**: (private, static, Scanner)
    - A static Scanner instance used for reading user input from the console.

Methods:

1. **main(String[] args)**
    - **Access Modifier**: public, static
    - **Return Type**: void
    - **Parameters**:
        - args (String[]): Command-line arguments.
    - **Description**:
        - Implements the main game loop.
        - Displays a menu allowing the user to start a new game, load a saved game, design or customize a board, or exit.
        - Executes actions based on the user's menu choice.

2. **startNewGame()**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Description**:
        - Prompts the user to enter the number of players.
        - Allows the user to input names for the players or generate random names.
        - Creates a new Game instance and passes it to the playGame method.

3. **loadGame()**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Description**:
        - Prompts the user to input the file name of a saved game.
        - Attempts to load the game using Game.loadGame(fileName).
        - If successful, calls playGame with the loaded Game instance.
        - Handles exceptions for loading errors.
    - **Exceptions**:
        - Catches IOException and ClassNotFoundException.

4. **playGame(Game game)**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Parameters**:
        - game (Game): The Game instance to play.
    - **Description**:
        - Implements the main gameplay loop for a single session.
        - Displays game status and allows the user to:
            - Roll dice and take turns.
            - View all player statuses.
            - View the next player in turn.
            - Save the game to a file.
            - Exit to the main menu.
        - Calls Game methods to handle gameplay mechanics.
        - Ends with the announcement of the winner.

5. **designNewBoard()**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Description**:
        - Creates a new GameBoard instance with default squares.
        - Calls customizeBoard to allow the user to modify the board.

6. **customizeExistingBoard()**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Description**:
        - Prompts the user to input the file name of a saved board.
        - Attempts to load the board using GameBoard.loadBoard(fileName).
        - Calls customizeBoard for further modifications if the board loads successfully.
        - Handles exceptions for loading errors.
    - **Exceptions**:
        - Catches IOException and ClassNotFoundException.

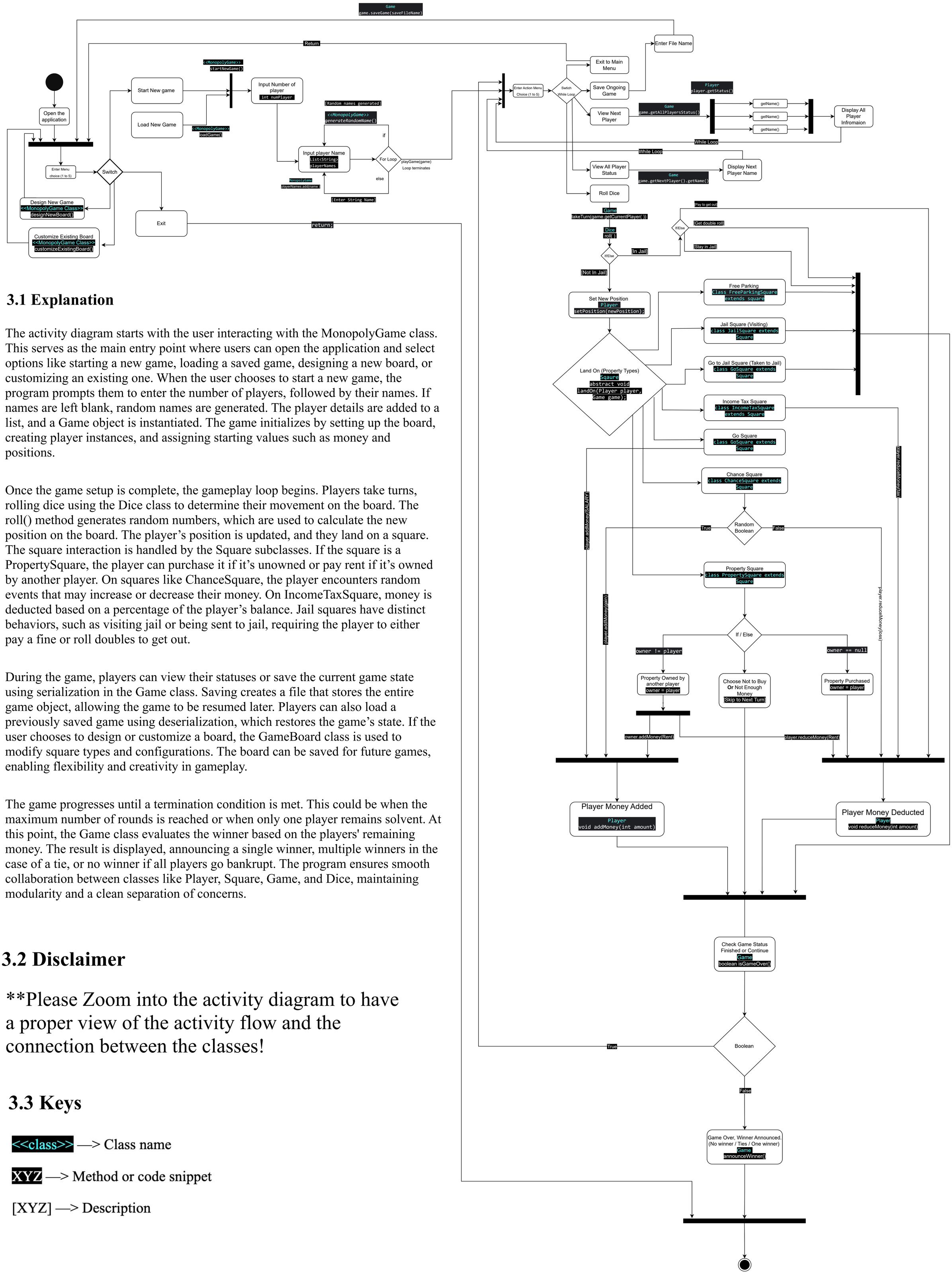
7. **customizeBoard(GameBoard board)**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Parameters**:
        - board (GameBoard): The board to customize.
    - **Description**:
        - Allows the user to modify squares, save the board, or return to the main menu.
        - Calls modifySquare for square customization.

8. **modifySquare(GameBoard board)**
    - **Access Modifier**: private, static
    - **Return Type**: void
    - **Parameters**:
        - board (GameBoard): The board to modify.
    - **Description**:
        - Prompts the user for the position of the square to modify and its new name.
        - Allows the user to select the type of square and enter additional details if required (e.g., price and rent for a PropertySquare).
        - Updates the square in the GameBoard using setSquare.

- **Exceptions**:
  - Handles invalid square types and inputs gracefully.

9. **generateRandomName()**
   - **Access Modifier**: private, static
   - **Return Type**: String
   - **Description**:
     - Generates a random name by combining a random adjective and a random noun from predefined arrays.
     - Returns the generated name.

3. Activity diagram



3.1 Explanation

The activity diagram starts with the user interacting with the MonopolyGame class. This serves as the main entry point where users can open the application and select options like starting a new game, loading a saved game, designing a new board, or customizing an existing one. When the user chooses to start a new game, the program prompts them to enter the number of players, followed by their names. If names are left blank, random names are generated. The player details are added to a list, and a Game object is instantiated. The game initializes by setting up the board, creating player instances, and assigning starting values such as money and positions.

Once the game setup is complete, the gameplay loop begins. Players take turns, rolling dice using the Dice class to determine their movement on the board. The roll() method generates random numbers, which are used to calculate the new position on the board. The player's position is updated, and they land on a square. The square interaction is handled by the Square subclasses. If the square is a PropertySquare, the player can purchase it if it's unowned or pay rent if it's owned by another player. On squares like ChanceSquare, the player encounters random events that may increase or decrease their money. On IncomeTaxSquare, money is deducted based on a percentage of the player's balance. Jail squares have distinct behaviors, such as visiting jail or being sent to jail, requiring the player to either pay a fine or roll doubles to get out.

During the game, players can view their statuses or save the current game state using serialization in the Game class. Saving creates a file that stores the entire game object, allowing the game to be resumed later. Players can also load a previously saved game using deserialization, which restores the game's state. If the user chooses to design or customize a board, the GameBoard class is used to modify square types and configurations. The board can be saved for future games, enabling flexibility and creativity in gameplay.

The game progresses until a termination condition is met. This could be when the maximum number of rounds is reached or when only one player remains solvent. At this point, the Game class evaluates the winner based on the players' remaining money. The result is displayed, announcing a single winner, multiple winners in the case of a tie, or no winner if all players go bankrupt. The program ensures smooth collaboration between classes like Player, Square, Game, and Dice, maintaining modularity and a clean separation of concerns.

3.2 Disclaimer

**Please Zoom into the activity diagram to have a proper view of the activity flow and the connection between the classes!

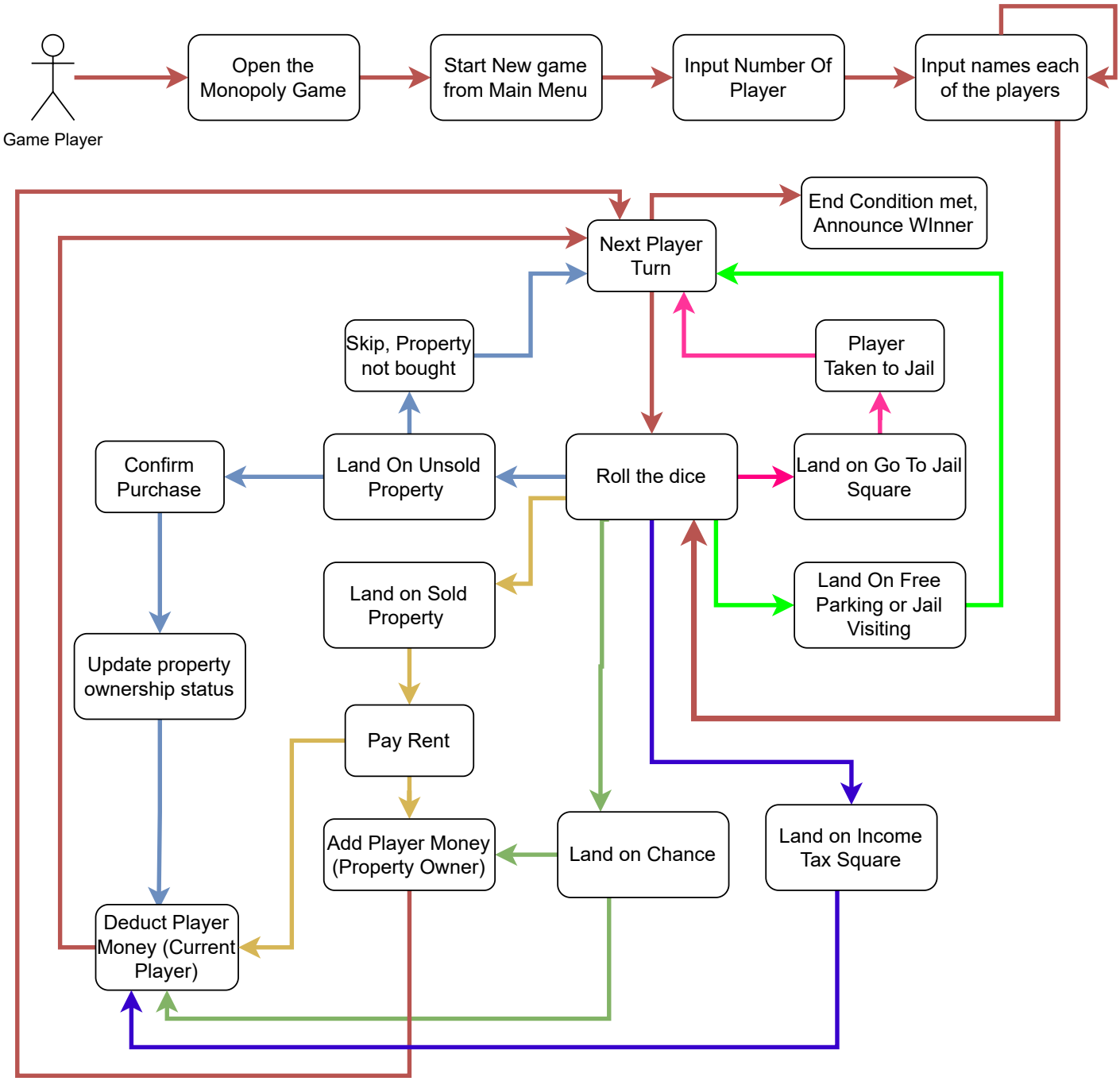
3.3 Keys

<<class>> —> Class name

XYZ —> Method or code snippet

[XYZ] —> Description

4. Game Play Diagram



Arrow Color

Red	Dark Blue	Dark Green	Yellow	Blue	Green	Pink

4.1 Game Play Diagram Description

★ **Starting the Turn:** The player begins their turn by **rolling the dice** to determine how many spaces they move on the board.

★ **Possible Actions Based on the Board Location:** Once the dice are rolled, the player moves accordingly and takes actions based on where they land:

1. **Land on Unsold Property**:
    - The player has two options:
        - **Buy the Property**:
            - The player purchases the property, updating its ownership status.
            - The property cost is deducted from the player's balance.
        - **Skip the Purchase**:
            - The property remains unsold, and no further action is taken.

2. **Land on Sold Property**:
    - The player must pay rent to the property's owner:
        - The rent amount is deducted from the player's money.
        - The same amount is added to the owner’s balance.

3. **Land on a Chance Square**:
    - The player draws a "Chance" card, which might involve:
        - Paying or receiving money.
        - Moving to a specific square on the board.
        - Other random effects determined by the card.

4. **Land on Income Tax Square**:
    - The player pays an income tax amount, which is deducted from their money balance.

5. **Land on Free Parking or Jail Visiting Square**:
    - No specific action is required. These spaces are neutral with no monetary impact.

6. **Land on "Go to Jail" Square**:
    - The player is sent directly to jail.
    - They must follow the jail rules in subsequent turns, which may include:
        - Paying a fine to leave jail.
        - Rolling doubles to get out.

★ **Ending the Turn:** After completing all actions resulting from their board location, the player's turn ends. The game proceeds to the **next player's turn**.

★ **Game Continuation and End Condition:** The game continues as players take turns rolling the dice, moving on the board, and interacting with spaces.

1. **End of the Game**:
    - The game ends when a pre-defined condition is met (e.g., a player goes bankrupt, 100 rounds completed)
    - Once the end condition is triggered, the winner is announced.

Color Legend for Diagram Arrows

- **Red**: Represents the main flow of a player's actions during their turn.
- **Dark Blue**: Highlights actions triggered by landing on "Income Tax" squares.
- **Yellow**: Indicates actions when the player lands on pre-owned property
- **Dark Green**: Highlights actions triggered by landing on "Chance" squares.
- **Blue**: Indicates actions when the player lands on unowned property
- **Green**: Covers neutral spaces like Free Parking or jail visits.
- **Pink**: Represents the action of being sent to jail after landing on the "Go to Jail" square.