FIT3077 Team 15
Sprint Two
Design Rationales

Key class:

During the design process, we debated two key classes within our team: the Move class and the Game class. We decided to create separate classes rather than incorporate their functionality as methods within other classes to ensure better separation of concerns and maintainability.

a. Move: The Move class was created to store attributes and functions related to piece movement. Initially, we considered incorporating this functionality into the Player or Piece classes, but that would make them "god classes," taking on too many tasks. Therefore, we created the Move class with attributes like startPosition, targetPosition, and playerType and a move function that utilizes these attributes to move the pieces.
b. Game: The Game class manages the overall game flow, rules, and state transitions. It contains functions that create Player1 and Player2, create the Board, and start the game. It checks whether the game is playing, handles move functions for the players, determines the current player's turn, and initializes the Board to place both players' pieces.

Key relations:

a. Game - Player: The relationship between the Game and Player classes is an aggregation, as the Game class contains references to the two Player objects but does not control their lifecycles. The Game class uses the Player objects to manage players' information and game state, while the Player class stores and maintains player-specific information.
b. Board - Piece: The relationship between the Board and Piece classes is a composition, as the Board class contains an array of Piece objects, and the Piece objects cannot exist without a Board object. The Board class is responsible for managing the game board state, while the Piece class represents the individual pieces on the board.

Decisions of inheritance:

In our design, we decided to use inheritance when designing the UI components. For example, in the BoardPanel class, we extended it from the JPanel class to create the game board. Similarly, in the MorrisGUI class, we extended the JFrame class to create the game UI. This inheritance allows us to leverage the existing functionality provided by the Java standard library.

Two Cardinalities:

We used the properties of different classes and their relationships to determine cardinalities. The primary criterion for determining cardinalities is the relationship between classes. For example, the cardinalities of the Player and Piece classes: In Nine Men's Morris, each player has 9 pieces, and each piece belongs to only one player. Players can also have no pieces at certain stages of the game. Therefore, the cardinality from Player to Piece is 0..9, and from Piece to Player is 1..1.

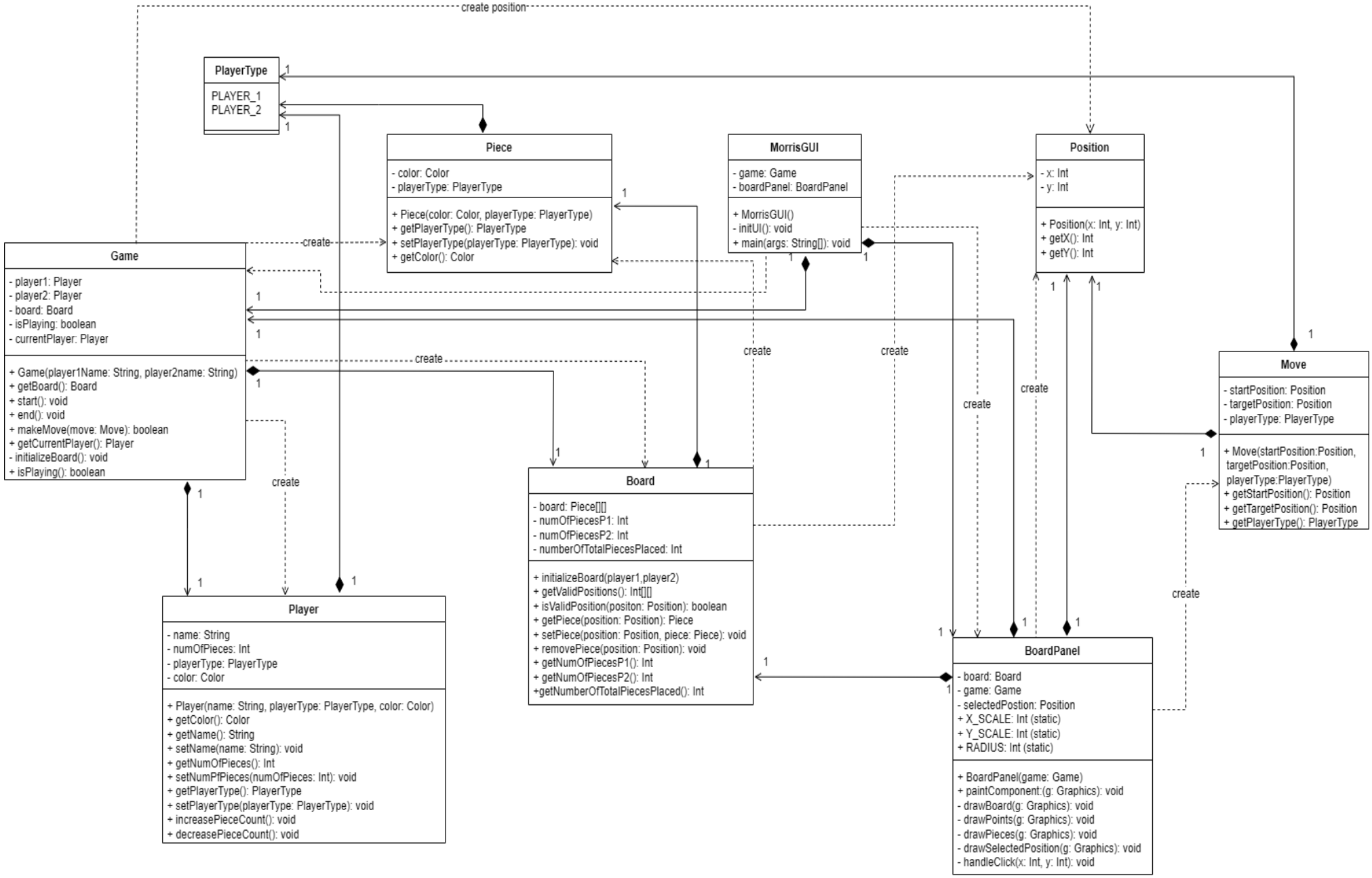
Design Patterns:

In our code, the design patterns used are not explicitly mentioned. However, the separation of concerns in the code can be considered an application of the Model-View-Controller (MVC) pattern. The Game class represents the Model, containing the game logic and state, the BoardPanel class represents the View, displaying the game state and handling user input, and the MorrisGUI class acts as the Controller, managing the user interface and game flow.

Two other feasible alternatives that were ruled out could be:
a. Observer Pattern: The Observer pattern could have been used to update the BoardPanel whenever the Game state changed. However, the current implementation opted for a simpler approach by directly calling the repaint() method in the BoardPanel class when necessary.

b. Factory Pattern: The Factory pattern could have been applied to create Piece objects for different player types. However, the current implementation is simple, and Piece objects are directly created within the Game class during board initialization, without the need for a separate factory.

Class Diagram:


create position
PlayerType
PLAYER_1
PLAYER_2
Piece
- color: Color
- playerType: PlayerType
+ Piece(color: Color, playerType: PlayerType)
+ getPlayerType(): PlayerType
+ setPlayerType(playerType: PlayerType): void
+ getColor(): Color
MorrisGUI
- game: Game
- boardPanel: BoardPanel
+ MorrisGUI()
- initUI(): void
+ main(args: String[]): void
Position
- x: Int
- y: Int
+ Position(x: Int, y: Int)
+ getX(): Int
+ getY(): Int
Game
- player1: Player
- player2: Player
- board: Board
- isPlaying: boolean
- currentPlayer: Player
+ Game(player1Name: String, player2name: String)
+ getBoard(): Board
+ start(): void
+ end(): void
+ makeMove(move: Move): boolean
+ getCurrentPlayer(): Player
- initializeBoard(): void
+ isPlaying(): boolean
Move
- startPosition: Position
- targetPosition: Position
- playerType: PlayerType
+ Move(startPosition:Position,
targetPosition:Position,
playerType:PlayerType)
+ getStartPosition(): Position
+ getTargetPosition(): Position
+ getPlayerType(): PlayerType
Board
- board: Piece[][]
- numOfPiecesP1: Int
- numOfPiecesP2: Int
- numberOfTotalPiecesPlaced: Int
+ initializeBoard(player1,player2)
+ getValidPositions(): Int[][]
+ isValidPosition(positon: Position): boolean
+ getPiece(position: Position): Piece
+ setPiece(position: Position, piece: Piece): void
+ removePiece(position: Position): void
+ getNumOfPiecesP1(): Int
+ getNumOfPiecesP2(): Int
+getNumberOfTotalPiecesPlaced(): Int
Player
- name: String
- numOfPieces: Int
- playerType: PlayerType
- color: Color
+ Player(name: String, playerType: PlayerType, color: Color)
+ getColor(): Color
+ getName(): String
+ setName(name: String): void
+ getNumOfPieces(): Int
+ setNumPfPieces(numOfPieces: Int): void
+ getPlayerType(): PlayerType
+ setPlayerType(playerType: PlayerType): void
+ increasePieceCount(): void
+ decreasePieceCount(): void
BoardPanel
- board: Board
- game: Game
- selectedPostion: Position
+ X_SCALE: Int (static)
+ Y_SCALE: Int (static)
+ RADIUS: Int (static)
+ BoardPanel(game: Game)
+ paintComponent:(g: Graphics): void
- drawBoard(g: Graphics): void
- drawPoints(g: Graphics): void
- drawPieces(g: Graphics): void
- drawSelectedPosition(g: Graphics): void
- handleClick(x: Int, y: Int): void
create
1

Program operation logic:

The primary Graphical User Interface (GUI) class, MorrisGUI, inherits from JFrame and is responsible for generating the game interface. Within the constructor, the game object, board panel, info text area are initialized, and the window's size, title, and other properties are set. A component adapter is included to maintain the square shape of the board.

The method initUI initializes User Interface components such as checkerboard panels, control panels, and information text areas. Within the control panel, the "End" button serves to terminate the game.The method updateInfoText refreshes the information text area to exhibit the current round number, player name, player color, and other relevant details.

BoardPanel, a custom JPanel class, is employed to render the game board and pieces. It comprises a Game object, a Board object, and a selected chess piece position, along with a mouse listener to manage user click events.

The method paintComponent is accountable for illustrating the board, points on the board, and pieces. When a pawn position is chosen, a red border is drawn to emphasize the selected position.

The handleClick method attends to user click events. It first ascertains the location of the user's click on the board, followed by updating the game state based on the selected piece's position and the clicked position.

The Board class embodies the game board's state, containing a 2D array to store pieces and variables to monitor the number of pieces for Player 1 and Player 2. Furthermore, it offers methods for obtaining, setting, and removing pieces, as well as verifying the validity of a given position.

The Game class signifies the game's logic, encompassing two Player objects, a Board object, a boolean indicating the game's progress, a Player object indicating the current player, and a variable counting the game rounds. Additionally, it provides methods for initiating the game, acquiring the current player, and executing game logic.

Upon starting the game, the pieces on the board are arranged in designated positions based on the player type. Users can interact with the game by selecting pieces and target positions. After each move, the program refreshes the info text area and examines the game state.