FIT3077 Team 15
Sprint Three
Design Rationales

Three Whys in each Class:

Modifications were made to the architecture primarily to enhance the modularity, readability, and maintainability of the code. New classes were introduced, such as Move and Mill, which independently handle functionalities that were previously scattered across other classes, such as movement and position checking. This makes the code logic more clear and organized. The majority of the code updates were made to accommodate new features and requirements.

In the new version of the Board class: Two new member variables have been added: "mills” and "adjacentPositions". "mills" represents the formed mills in the game, which are lines on the board where three pieces are aligned. "adjacentPositions" represents the adjacent positions for each position on the game board.

In the previous version of the Board class, these two member variables did not exist.

2. A new static constant array named VALID_POSITIONS has been defined to store all the valid positions on the game board.

3. In the initialization method (constructor), in addition to initializing the member variables, a new method initializeAdjacentPositions() is called to initialize the adjacent positions for each position on the game board. For each valid position, a list containing all valid adjacent positions is created.

This operation was not performed in the previous version of the Board class.

4. In the new version of the Board class, several new or updated methods have been added, such as:

isValidPosition(): checks if a position is valid for placing a piece during gameplay.

getPiece(): retrieves the specific position of a piece on the board.

setPiece(): places a piece at a specified coordinate on the board.

movePiece(): moves a piece from one position to another.

removePiece(): removes a piece from the board.

formMill(): checks and identifies if a mill is formed on the board.

isMill(): checks if a mill is currently formed on a specific position.

isValidMove(): checks if a piece movement is legal. In the moving phase, the move is valid if the destination position is adjacent to the starting position. In the flying phase, any move is valid as long as the destination is empty.

getOpponentNonMillPieces(): retrieves the opponent's non-mill pieces on the board.

isInMill(): checks if a specific piece is part of a mill.

getPossibleMills(): returns a list of possible mills on the current board configuration.

The code adheres to certain design principles, such as encapsulation (using getter and setter methods to access and modify data) and modularity (implementing different functionalities as independent methods). It also consciously reduces the public interface of the class, which are good object-oriented design practices.

Added functionalities: The new version of the Board class introduces numerous new functionalities that were not present in the old version. These include checking the validity of a position, getting and setting pieces, moving pieces, removing pieces, forming mills, checking if a mill is formed, validating the legality of a move, and retrieving opponent's non-mill pieces. These functionalities support the new game rules and logic.

Changes made: The changes made in the new version primarily involve the addition of new member variables and methods to support the new game rules and logic, such as mill formation and checking, validation of legal moves, and more. Additionally, modifications were made to certain existing methods (such as the constructor) to accommodate the new member variables and game logic.

In the new version of Game class: Revised Architecture: The architecture was revised to accommodate new game rules and provide a more modular and extensible design. The changes include the addition of game phases, tracking player pieces to be placed and pieces left, and introducing player-specific phases.(Considering the limitations of space and font size, I will not provide specific examples.) These changes improve the game's flexibility, making it easier to modify or add new features in the future.

Quality Attribute:
Usability: Usability: is crucial for 9MM games to ensure that players can understand and interact with the game. The revised design combines clear player turns, intuitive piece placement, and verification of legal moves.

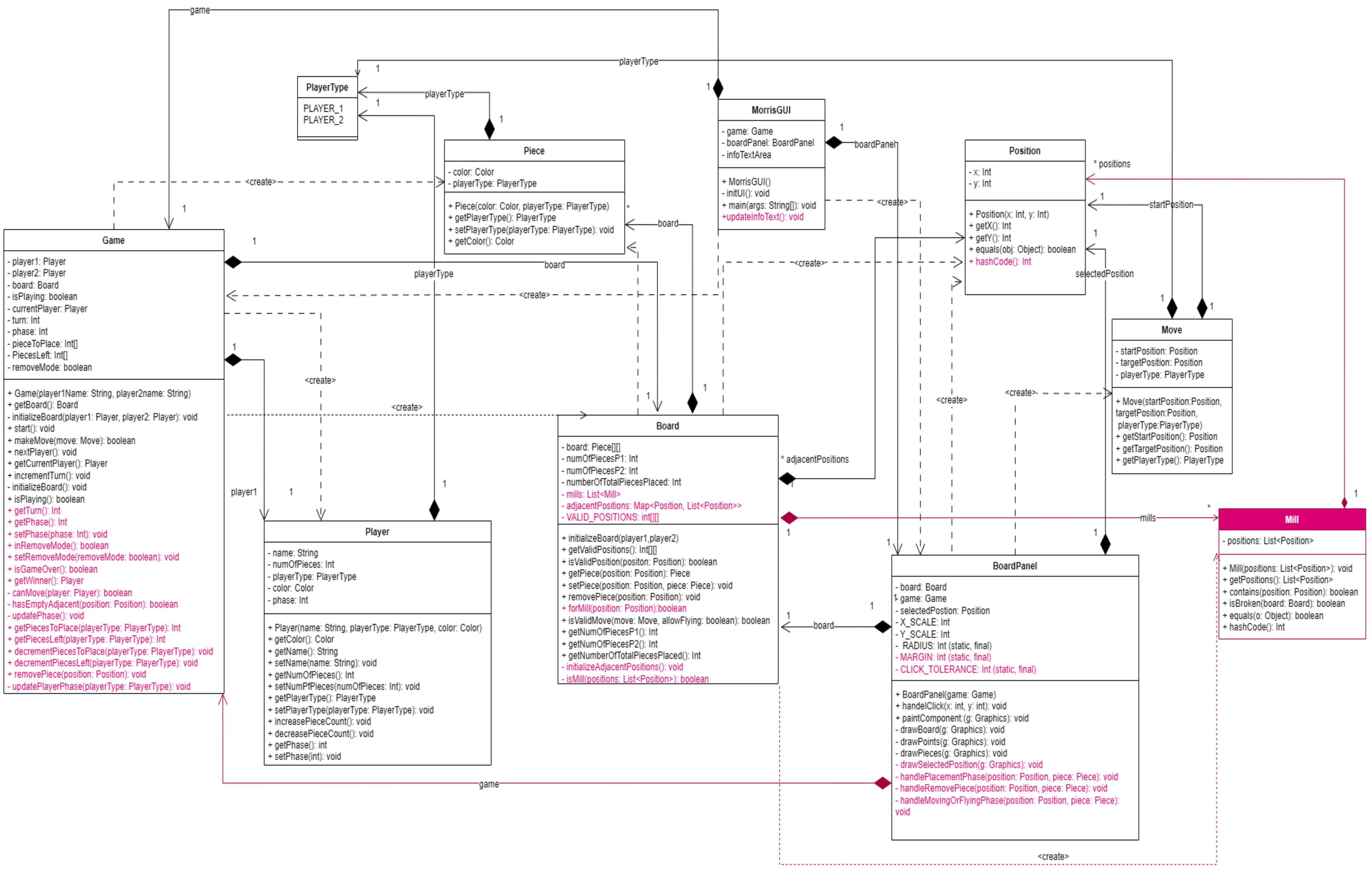
Maintainability: The revised design introduced modularity, encapsulation, and separation of concerns. It includes separate classes for game logic (Game), player information (Player), and game board representation (Board). These design choices improve code organization, readability, and facilitate future enhancements or bug fixes. The new design improves code maintainability by creating new classes and decomposing functionalities. For example, the Mill class independently handles issues related to board positions, while the Move class handles the movement of pieces. This decomposition makes the code more clear and facilitates easier maintenance and modifications.

Flexibility:The new design takes into consideration the potential expansion requirements of the game. For instance, the addition of the phase property in the Player class enables the game to incorporate additional states and rules. This design facilitates the scalability of the game, allowing for easy expansion and the incorporation of new features.

Stimulation: Stimulation is the ability for players to freely make choices and take actions within the game. In the design, players are given the freedom to move their pieces on the board and place them in possible positions during special phases. This is reflected in classes such as Board, Position, and Move. The Board class represents the game board, the Position class represents a specific position on the board, and the Move class represents a single movement from one position to another. These designs provide players with the space to exercise their agency, allowing them to strategically select optimal moves and placements based on their own strategies.

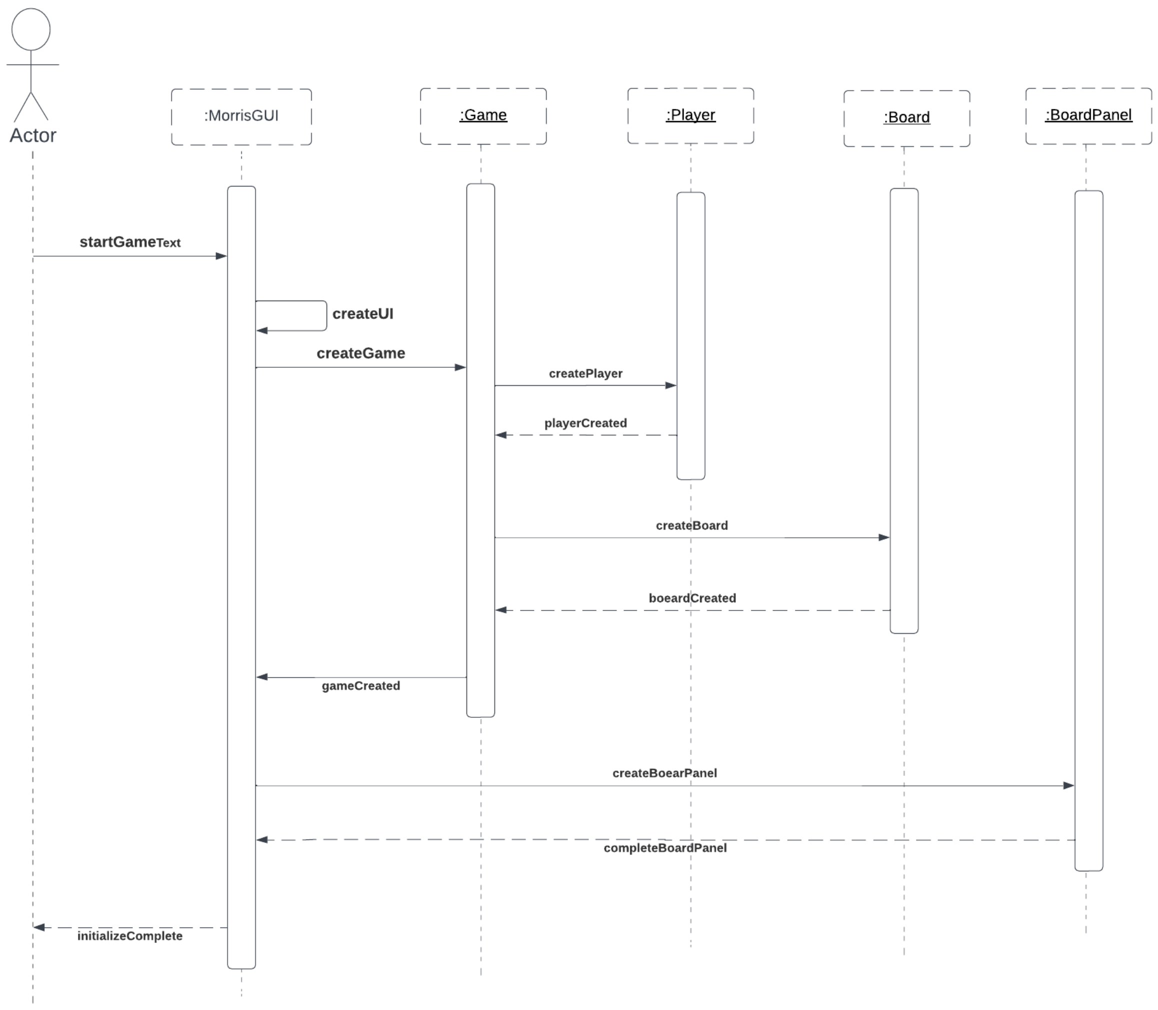
Achievement: Achievement is a core experience that players seek in a game, as it provides a strong sense of satisfaction and fulfillment when players achieve victory through strategic thinking and decision-making. In the design, the game rules and logic are clear and well-defined, allowing players to influence the game outcome through their own efforts and skills. This is reflected in the classes it has created, such as Player, Move, and Mill, which are designed around player actions and game results. For instance, the Player class contains essential player information and current status, the Move class represents a single player's move, and the Mill class is used to determine whether a player has formed a "mill," which refers to aligning three pieces in a row. These designs revolve around player achievement, providing ample opportunities for players to showcase their abilities.

Revised Class Diagram:

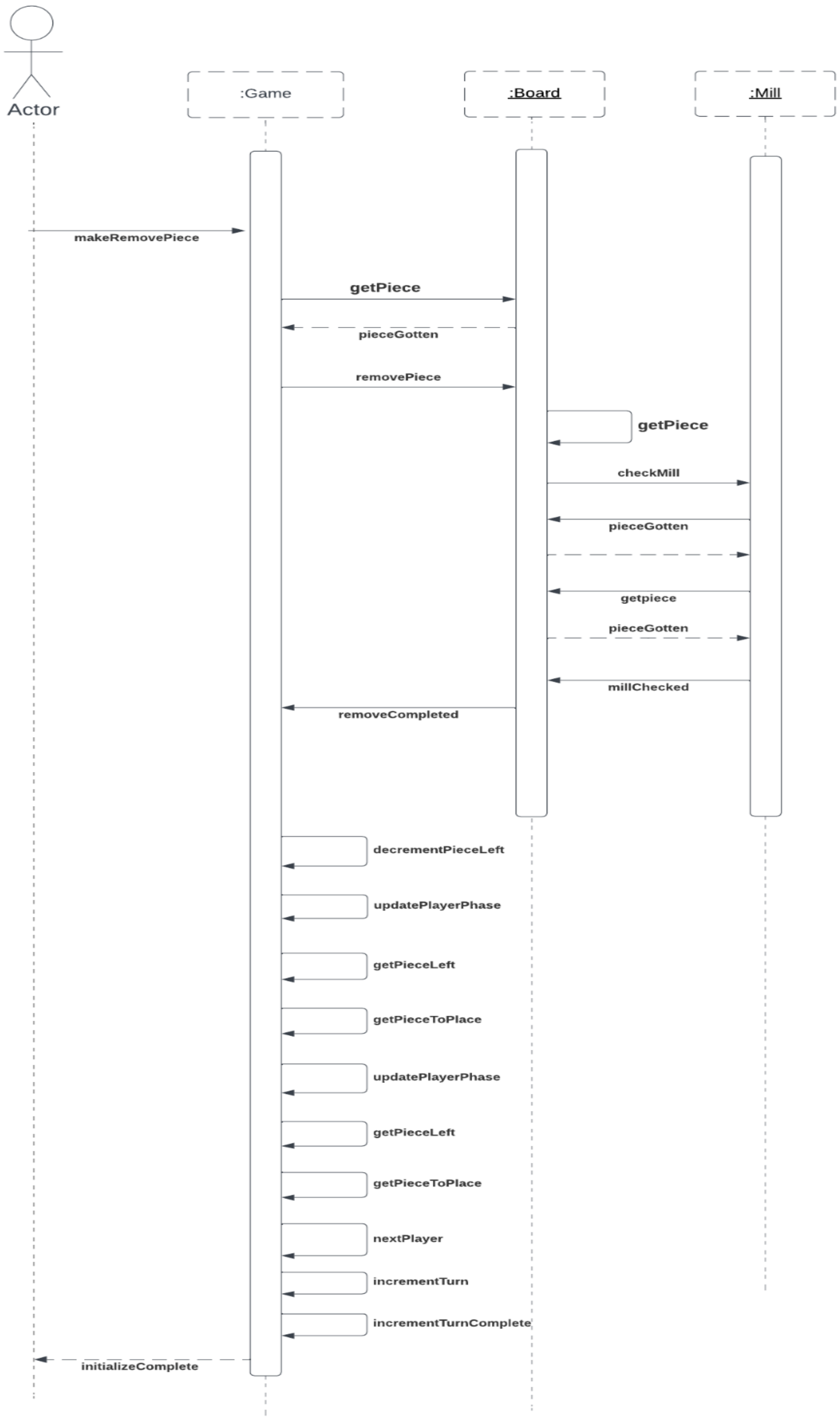

game
playerType
PlayerType
PLAYER_1
PLAYER_2
playerType
Piece
- color: Color
- playerType: PlayerType
+ Piece(color: Color, playerType: PlayerType)
+ getPlayerType(): PlayerType
+ setPlayerType(playerType: PlayerType): void
+ getColor(): Color
MorrisGUI
- game: Game
- boardPanel: BoardPanel
- infoTextArea
+ MorrisGUI()
- initUI(): void
+ main(args: String[]): void
+updateInfoText(): void
boardPanel
Position
- x: Int
- y: Int
+ Position(x: Int, y: Int)
+ getX(): Int
+ getY(): Int
+ equals(obj: Object): boolean
+ hashCode(): Int
* positions
startPosition
selectedPosition
<create>
board
Game
- player1: Player
- player2: Player
- board: Board
- isPlaying: boolean
- currentPlayer: Player
- turn: Int
- phase: Int
- pieceToPlace: Int[]
- PiecesLeft: Int[]
- removeMode: boolean
+ Game(player1Name: String, player2name: String)
+ getBoard(): Board
- initializeBoard(player1: Player, player2: Player): void
+ start(): void
+ makeMove(move: Move): boolean
+ nextPlayer(): void
+ getCurrentPlayer(): Player
+ incrementTurn(): void
- initializeBoard(): void
+ isPlaying(): boolean
+ getTurn(): Int
+ getPhase(): Int
+ setPhase(phase: Int): void
+ inRemoveMode(): boolean
+ setRemoveMode(removeMode: boolean): void
+ isGameOver(): boolean
+ getWinner(): Player
- canMove(player: Player): boolean
- hasEmptyAdjacent(position: Position): boolean
- updatePhase(): void
+ getPiecesToPlace(playerType: PlayerType): Int
+ getPiecesLeft(playerType: PlayerType): Int
+ decrementPiecesToPlace(playerType: PlayerType): void
+ decrementPiecesLeft(playerType: PlayerType): void
+ removePiece(position: Position): void
- updatePlayerPhase(playerType: PlayerType): void
player1
Player
- name: String
- numOfPieces: Int
- playerType: PlayerType
- color: Color
- phase: Int
+ 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
+ getPhase(): int
+ setPhase(int): void
Board
- board: Piece[][]
- numOfPiecesP1: Int
- numOfPiecesP2: Int
- numberOfTotalPiecesPlaced: Int
- mills: List<Mill>
- adjacentPositions: Map<Position, List<Position>>
- VALID_POSITIONS: int[][]
+ initializeBoard(player1,player2)
+ getValidPositions(): Int[][]
+ isValidPosition(positon: Position): boolean
+ getPiece(position: Position): Piece
+ setPiece(position: Position, piece: Piece): void
+ removePiece(position: Position): void
+ forMill(position: Position):boolean
+ isValidMove(move: Move, allowFlying: boolean): boolean
+ getNumOfPiecesP1(): Int
+ getNumOfPiecesP2(): Int
+ getNumberOfTotalPiecesPlaced(): Int
- initializeAdjacentPositions(): void
- isMill(positions: List<Position>): boolean
* adjacentPositions
Move
- startPosition: Position
- targetPosition: Position
- playerType: PlayerType
+ Move(startPosition:Position, targetPosition:Position, playerType:PlayerType)
+ getStartPosition(): Position
+ getTargetPosition(): Position
+ getPlayerType(): PlayerType
mills
Mill
- positions: List<Position>
+ Mill(positions: List<Position>): void
+ getPositions(): List<Position>
+ contains(position: Position): boolean
+ isBroken(board: Board): boolean
+ equals(o: Object): boolean
+ hashCode(): Int
BoardPanel
- board: Board
- game: Game
- selectedPostion: Position
- X_SCALE: Int
- Y_SCALE: Int
- RADIUS: Int (static, final)
- MARGIN: Int (static, final)
- CLICK_TOLERANCE: Int (static, final)
+ BoardPanel(game: Game)
+ handelClick(x: int, y: int): void
+ paintComponent:(g: Graphics): void
- drawBoard(g: Graphics): void
- drawPoints(g: Graphics): void
- drawPieces(g: Graphics): void
- drawSelectedPosition(g: Graphics): void
- handlePlacementPhase(position: Position, piece: Piece): void
- handleRemovePiece(position: Position, piece: Piece): void
- handleMovingOrFlyingPhase(position: Position, piece: Piece): void
game
<create>

Sequence or communication diagrams:

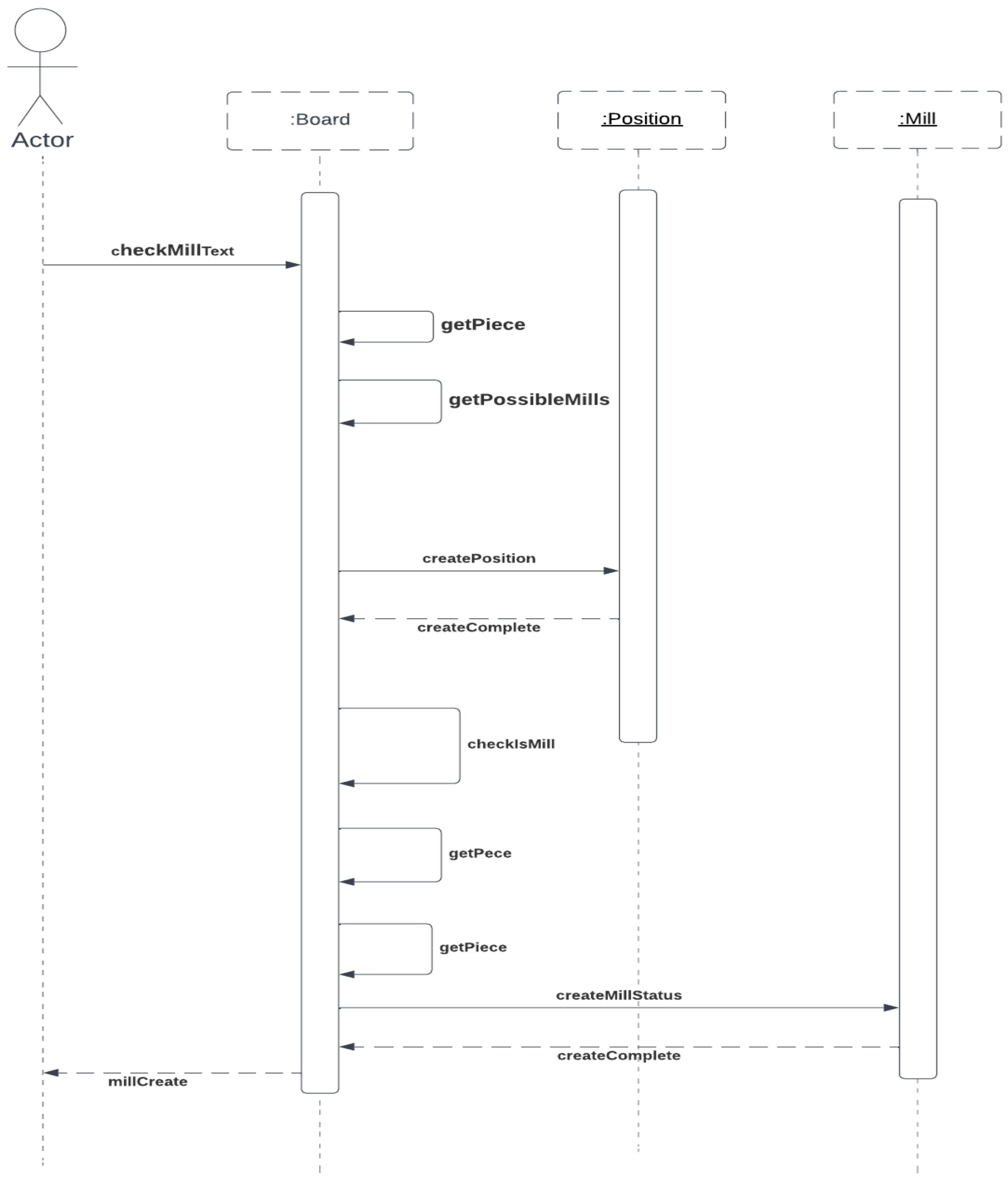
Initialisation process:



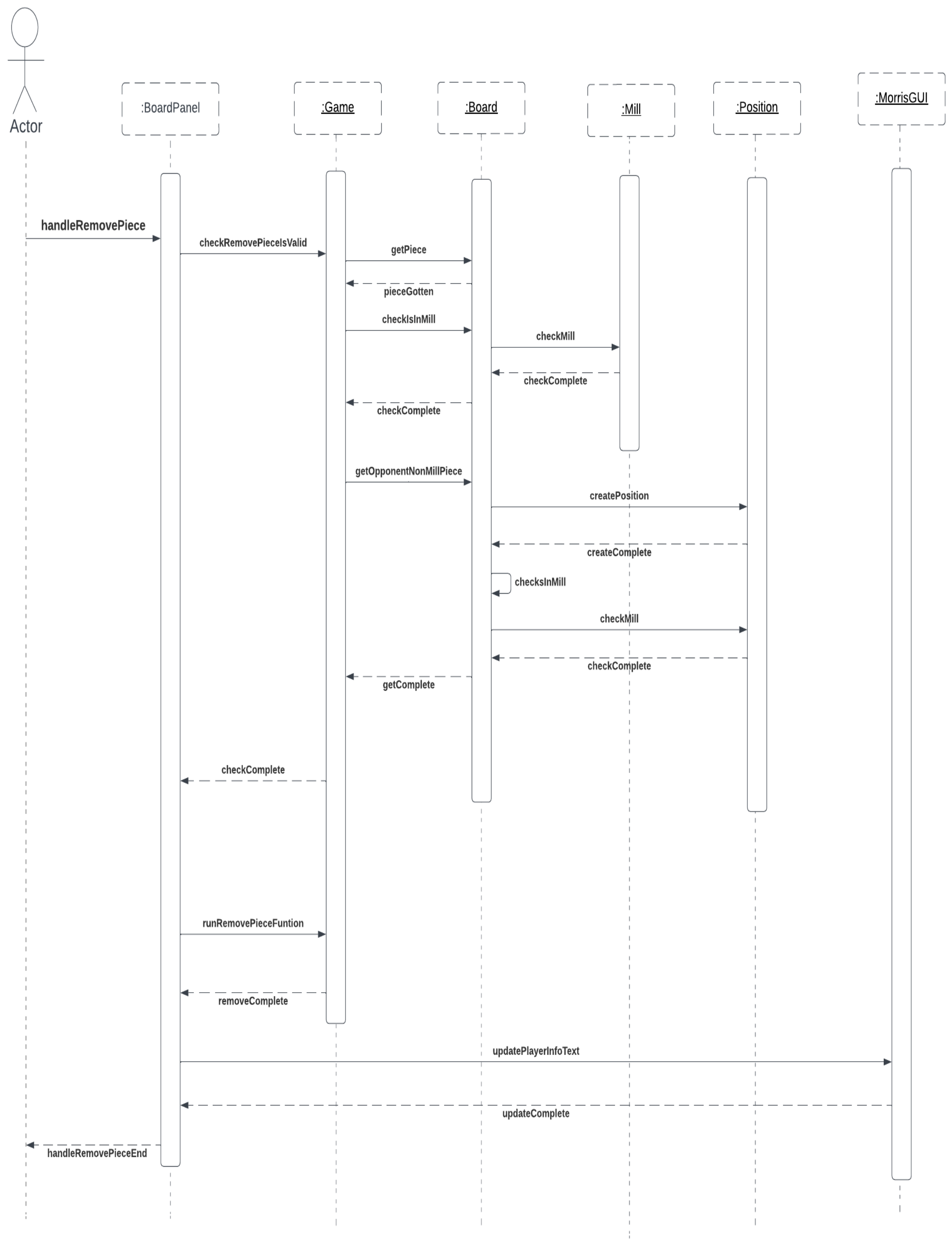
Remove Pieces:


Actor
:Game
:Board
:Mill
makeRemovePiece
getPiece
pieceGotten
removePiece
getPiece
checkMill
pieceGotten
getpiece
pieceGotten
millChecked
removeCompleted
decrementPieceLeft
updatePlayerPhase
getPieceLeft
getPieceToPlace
updatePlayerPhase
getPieceLeft
getPieceToPlace
nextPlayer
incrementTurn
incrementTurnComplete
initializeComplete

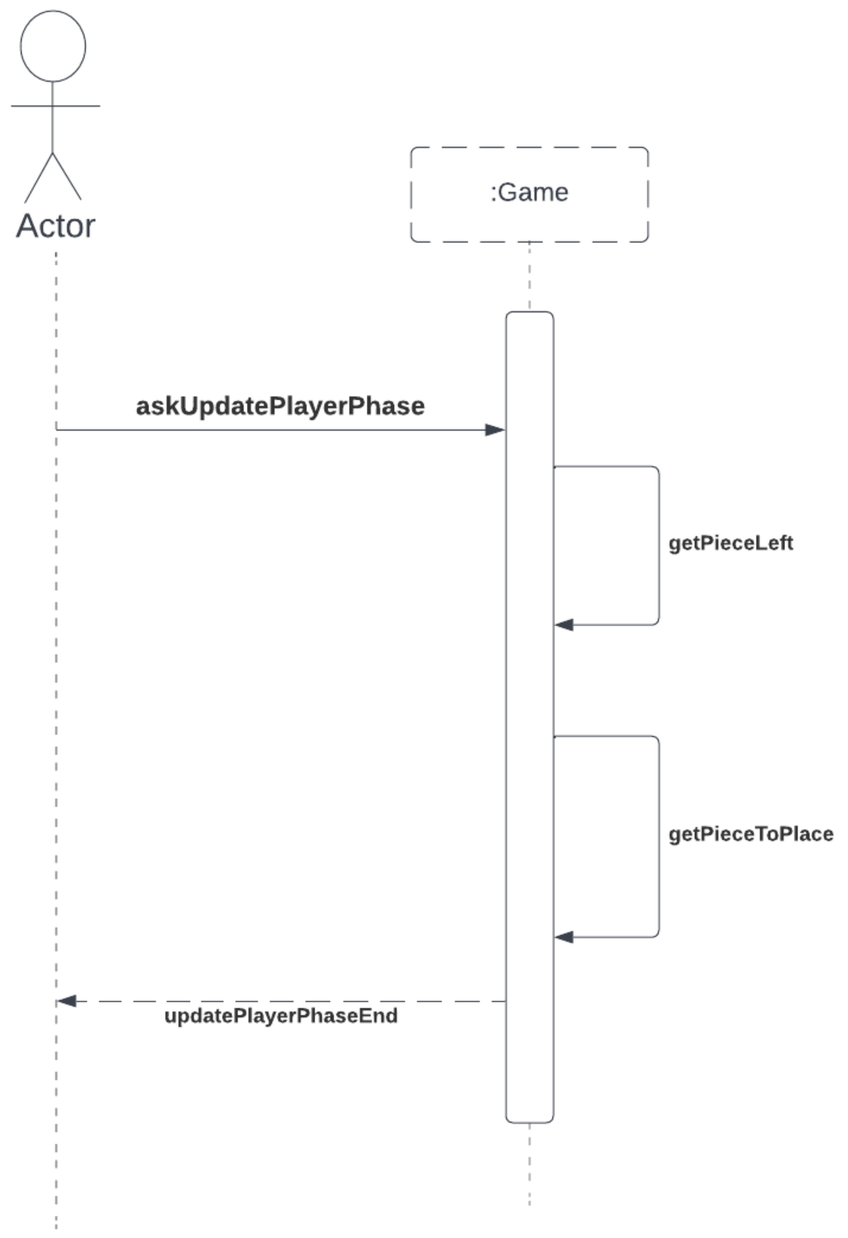
Form Mill:



Handle Remove Pieces:


Actor
:BoardPanel
:Game
:Board
:Mill
:Position
:MorrisGUI
handleRemovePiece
checkRemovePieceIsValid
getPiece
pieceGotten
checkIsInMill
checkMill
checkComplete
checkComplete
getOpponentNonMillPiece
createPosition
createComplete
checksInMill
checkMill
checkComplete
getComplete
checkComplete
runRemovePieceFuntion
removeComplete
updatePlayerInfoText
updateComplete
handleRemovePieceEnd

Update Player Phase:


Actor
:Game
askUpdatePlayerPhase
getPieceLeft
getPieceToPlace
updatePlayerPhaseEnd