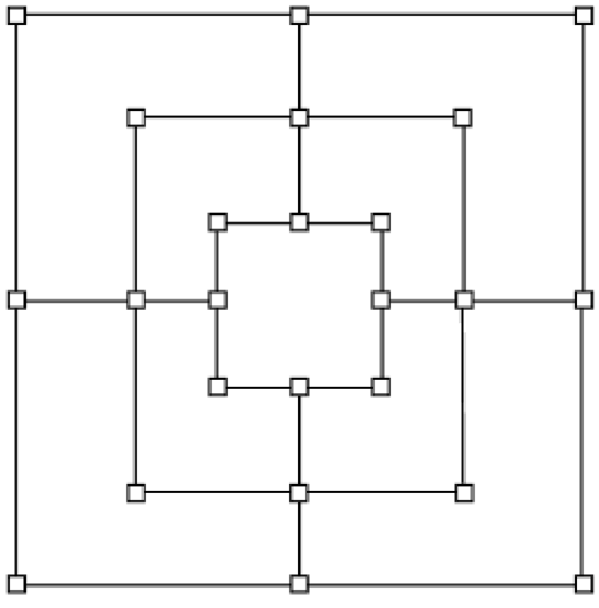
CIS*3260 Design: Nine Men's Morris

Designers:
Aaron McNeil
Andrew Carson
David Solus
Eric Morgan
Jason Chow
Jing Xuan Long



CIS*3260
Dr. Mark Wineberg
University of Guelph
2021/11/28

Table of Contents

Executive Summary

Game Description

This Design Document models a version of the ancient board game Nine Men’s Morris to be implemented as a web application running in ruby on rails. The goal of Nine Men’s Morris is to place your pieces on the board, form ‘mills’ of 3 pieces, and leave your opponent either with 2 pieces or 0 moves to win the game.

High Level Overview

For the implementation design of Nine Men’s Morris, we used **7 use cases** for the base of our design: Setup Game, Place Game Piece, Move Game Piece, Fly Game Piece, Capture Game Piece, End the Game, and Propose a Draw.

We created 7 classes: Referee, Game, Player, Bag, Piece, GameBoard, and Intersection. These 7 classes are used by the GameController within a rails system to achieve Nine Men's Morris gameplay. The rails frontend is also composed of 3 other controllers that exist to manage database operations and web browser functions .

Included are 14 client server sequence diagrams: Login, Load the Player Home, Logout of Account, Forfeit the Game, Propose a Draw, Play Again, Signup - Create an account, Login Player (happens when login button pressed), Move a Piece, Place a Piece, Capture a Piece, Delete User Account, and matchmaking. The sequence diagrams give a full top-bottom overview of system flow from rails to the game subsystem.

We have included designs for 9 webpages: Login Page, Sign up Page, Player Home Page, Matchmaking Page, Game Page (Placing Phase - Where the game starts), Game Page (Movement Phase), Game Page (After forming mill, page will display capture interface), GamePage (Draw accept/decline), Game Page (Game Finished).

Important Limitations and Things to Consider

When creating the backend, keep in mind that the coordinates for the pieces (the x and y position) are all integers. This is important to remember because in the data model for intersections, they are characters, however, to make it possible to use 2-D arrays, the characters that are sent from the client should always be converted to zero-indexed integers before being sent to the application subsystem. Please see the Gameboard constructor details for more information on how the intersection array operates.

Game Rules

All content in this section ‘Game Rules’ is taken directly from the provided reference.

Note: In this system's specific ruleset, white will always move first.

Reference

https://en.wikipedia.org/wiki/Nine_men's_morris

Background Info

Nine men’s morris is a strategy board game for two players dating back to the Roman Empire. It is also known as nine-man morris, mill, mills, the mill game, merels, merrills, merelles, marelles, morelles, ninepenny marl, and cowboy checkers. Nine men’s morris is a solved game, that is, a game whose optimal strategy has been calculated. With perfect play from both players, the game results in a draw.

Game Board and Pieces

The board consists of a grid with twenty-four *intersections* or *points*. Each player has nine pieces, or "men", usually coloured black and white. Players try to form 'mills'—three of their own men lined horizontally or vertically—allowing a player to remove an opponent's man from the game. A player wins by reducing the opponent to two pieces (where they could no longer form mills and thus be unable to win), or by leaving them without a legal move.

The game proceeds in three phases:

1. Placing men on vacant points
2. Moving men to adjacent points
3. (optional phase) Moving men to any vacant point (flying) when the player has been reduced to three men

Phase 1: Placing Pieces

The game begins with an empty board. The players determine who plays first, then take turns placing their men one per play on empty points. If a player is able to place three of their pieces on contiguous points in a straight line, vertically or horizontally, they have formed a *mill* and may remove one of their opponent's pieces from the board and the game, with the caveat that a piece in an opponent's mill can only be removed if no other pieces are available. After all men have been placed, phase two begins.

Phase 2: Moving Pieces

Players continue to alternate moves, this time moving a man to an adjacent point. A piece may not "jump" another piece. Players continue to try to form mills and remove their opponent's pieces as in phase one. A player can "break" a mill by moving one of his pieces out of an existing mill, then moving it back to form the same mill a second time (or any number of times), each time removing one of his opponent's men. The act of removing an opponent's man is sometimes called "pounding" the opponent. When one player has been reduced to three men, phase three begins.

Phase 3: “Flying”

When a player is reduced to three pieces, there is no longer a limitation on that player of moving to only adjacent points: The player's men may "fly" (or "hop", or "jump") from any point to any vacant point.

Valid Moves

Placing

During the placement phase, a player may select a piece from their bag and place it on the game board. If they manage to place 3 pieces adjacent to each other vertically or horizontally then they may capture an opposing piece.

Standard Move

During the movement phase when a player has more than 3 pieces on the board they may move any piece to any vertically, or horizontally adjacent intersection. A player may not move a piece diagonally.

Fly

When a player has exactly 3 pieces remaining they can execute a fly. This allows them to move any piece to any open intersection on the board (adjacency no longer matters).

Captures

After a mill has been created a player will be able to capture an enemy piece. Under normal circumstances a player is only allowed to capture an enemy piece that is not in a mill, however, if every single one of the enemy's pieces are in a mill the piece is allowed to be captured. Once a piece has been captured it will never re-enter the game.

General Goal

The goal of the game is to continue forming mills to capture enemy player pieces. Each time that you form a mill you can capture one piece. Once the enemy has 2 pieces remaining or no more valid moves, you have won the game.

Use Cases

Use Case 1

Use Case: Setup Game
Primary Actor: Player
Goal: To ensure players and the game board are ready for a game
Stakeholders List:

- Player
- Opponent
- Referee

Initiating Event:
Main Success Scenario:

1. Referee designates each player as white or black
2. Referee clears and sets up board
3. Player states that they are ready to start the game
4. Opponent states that they are ready to start the game

Post Conditions:

- Game has started
- White player is ready to make their first move

Alternate Flows or Exceptions: N/A

Use Cases Utilized: N/A
Scenario Notes:

Use Case 2

Use Case: Place game piece
Primary Actor: Player
Goal: A player wants to place one of their off-board game pieces on an empty intersection on the board.
Stakeholders List:

- Player - Wants to place a game piece on the board

Initiating Event: The game is in the first phase (placing phase) and it is the player's turn to place one of their pieces on the board.
Main Success Scenario:

1. Player selects one of their remaining pieces

2. Player states where they want to place their piece
3. Referee determines that the placement is valid
4. Player places game piece on the empty intersection on the board

Post Conditions:

- Player has one less game piece remaining off the board
- Game piece has been added to board

Alternate Flows or Exceptions:

1.a. Player runs out of pieces to place

1.a.1) Referee determines that the first phase of the game is complete (placing phase)

1.a.2) Referee tells both players that the second phase of the game has started

2.a. Referee determines that placement is invalid

2.a.1) Referee tells player that the placement is invalid

2.a.2) Player chooses a new intersection to place piece

Use Cases Utilized: N/A

Scenario Notes: Known as the placing phase. All 9 pieces must be placed by each player.

Use Case 3

Use Case: Move game piece
Primary Actor: Player
Goal: The player wants to move one of their game pieces from its current intersection to an adjacent intersection.
Stakeholders List:

- Player - Wants to move piece to an adjacent intersection
- Opponent
- Referee

Initiating Event: All pieces have been added to the board and the player's turn has begun.
Main Success Scenario:

1. Player chooses a game piece that they want to move
2. Player states what piece they want to move and where they want to move it to
3. Referee determines that the move is valid
4. Player moves game piece to an adjacent intersection

Post Conditions:

- Game piece has been moved to an empty adjacent intersection

- The intersection that the game piece used to occupy is now empty

Alternate Flows or Exceptions:

3.a. Referee determines that movement is invalid

3.a.1) Referee tells player that the move is invalid

3.a.2) Player chooses a different action to take

Use Cases Utilized: N/A

Scenario Notes: The player has more than 3 game pieces remaining on the board, so they can only move pieces to adjacent, empty intersections.

Use Case 4

Use Case: Fly game piece

Primary Actor: Player

Goal: The player wants to move one of their pieces to any non-adjacent, empty intersection on the board.

Stakeholders List:

- Player - Wants to move a game piece using a “Fly”
- Opponent
- Referee

Initiating Event: It’s the player’s turn and they only have 3 pieces remaining.

Main Success Scenario:

1. Player chooses a game piece that they want to move
2. Player states what piece they want to move and where they want to move it to
3. Referee determines that the move is valid
4. Player moves their game piece to an empty non-adjacent intersection

Post Conditions:

- Game piece has been moved to an empty intersection
- The intersection that the game piece used to occupy is now empty

Alternate Flows or Exceptions:

1.a. Referee determines that the action is invalid

1.a.1) Referee tell player that their action is invalid

1.a.2) Player chooses a new action to take

Use Cases Utilized: N/A

Scenario Notes: N/A

Use Case 5

Use Case: Capture a piece
Primary Actor: Player
Goal: The player wants to capture (remove) an enemy game piece.
Stakeholders List:

- Player - Wants to capture an enemy piece
- Opponent
- Referee

Initiating Event: Player has moved a piece or placed a piece resulting in them having 3 pieces in a row horizontally or vertically.
Main Success Scenario:

1. Referee determines that Player has formed a mill with their last move
2. Player states what piece they want to capture
3. Referee determines that the capture is valid
4. Player captures an opposing game piece

Post Conditions:

- Opposing game piece has been removed
- The intersection that the game piece occupied is now empty

Alternate Flows or Exceptions:

1.a. Referee determines that the capture is invalid
  1.a.1) Referee tells player that their capture is invalid
  1.a.2) Player chooses a difference piece capture

Use Cases Utilized: N/A

Scenario Notes: N/A

Use Case 6

Use Case: End the game
Primary Actor: Player
Goal: The referee determines if the move made by the player will end the game.
Stakeholders List:

- Player moving - Wants to win the game
- Opponent
- Referee - Determines whether the game has ended and who the victor is

Initiating Event: Player creates a mill and claims the opponent's 3rd last piece.
Main Success Scenario:
1. Player captures an Opponent piece
2. Referee verifies that the opponent only has two pieces left
3. Referee determines the winning player
4. Referee resets the game boards

Post Conditions:
- The game is over
- Winner is decided
- Players are prompted to play again

Alternate Flows or Exceptions:
1.a Referee determines that the opponent still has three or more pieces remaining
1.a.1) Referee determines that game is not in an end state
1.a.2) Referee tells both players that the game is not over

Use Cases Utilized:
Use Case 5

Scenario Notes: N/A

Use Case 7

Use Case: Propose Game Draw
Primary Actor: Player, Opponent Player
Goal: The player taking their turn wishes to propose a draw to determine if the game should end.
Stakeholders List:
- Player moving - Wants the game to be concluded with a draw
- Opposing player - Must accept or decline proposal to determine game state

Initiating Event: Both players have their pieces in a position in which they can infinitely prevent each other from forming a mill needed to win.
Main Success Scenario:
1. Player proposes a draw
2. Opponent Player accepts the draw
3. Referee declares that the game has ended in a draw

Post Conditions:
- The game is over
- No winner is decided

- Players are prompted to play again

Alternate Flows or Exceptions:

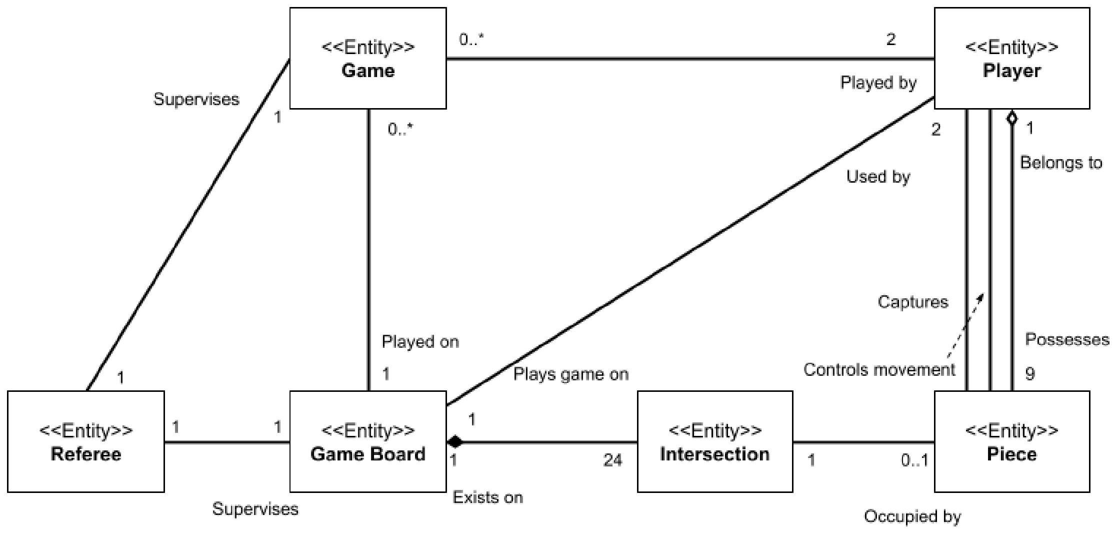
1.a. Opponent denies the draw proposal

1.a.1) Referee declares that the game will continue

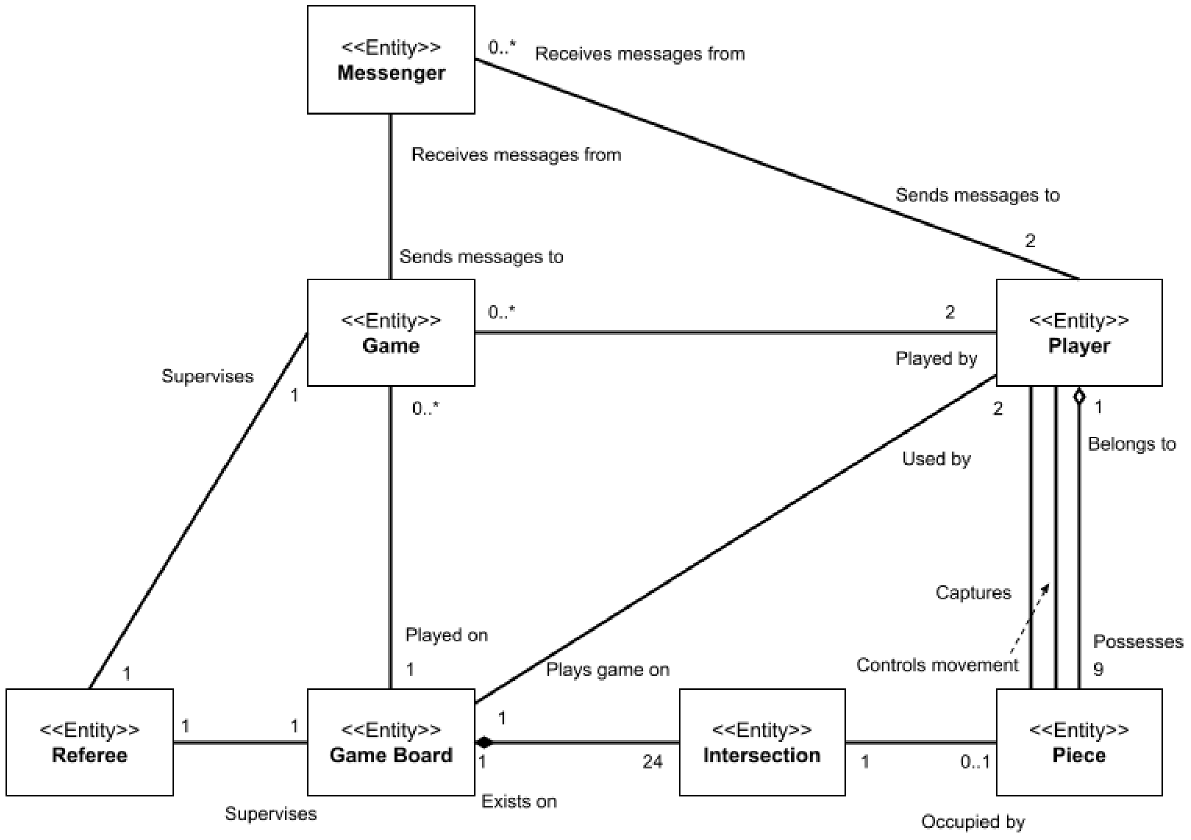
Use Cases Utilized: N/A

Scenario Notes: For the Draw case, it is difficult to determine a draw state, since both players can still move but may block each other from forming a mill infinitely. This case is impossible to implement algorithmically.

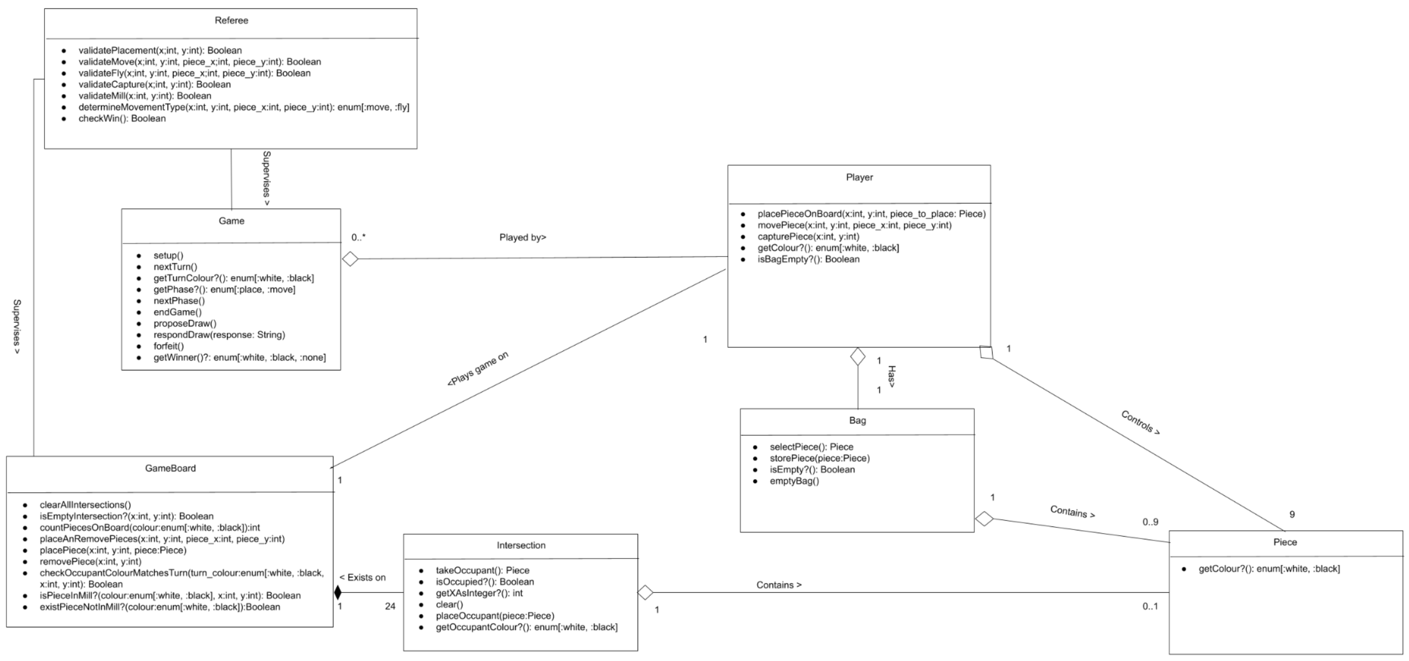
Entity Diagram



Remote Entity Diagram

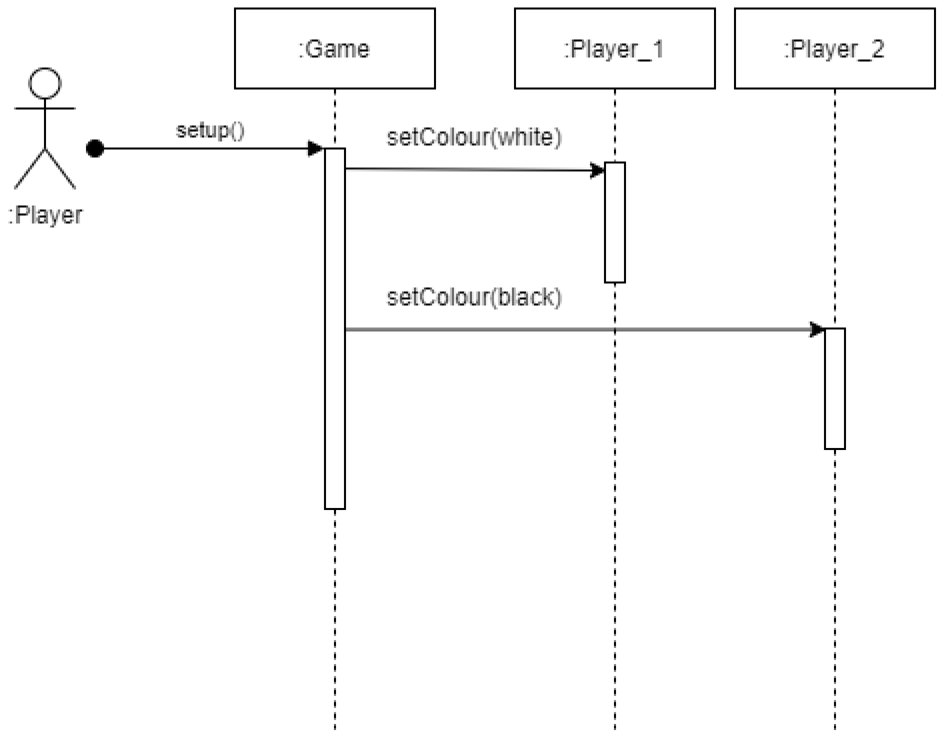


Object Interface Class Diagram (Application)

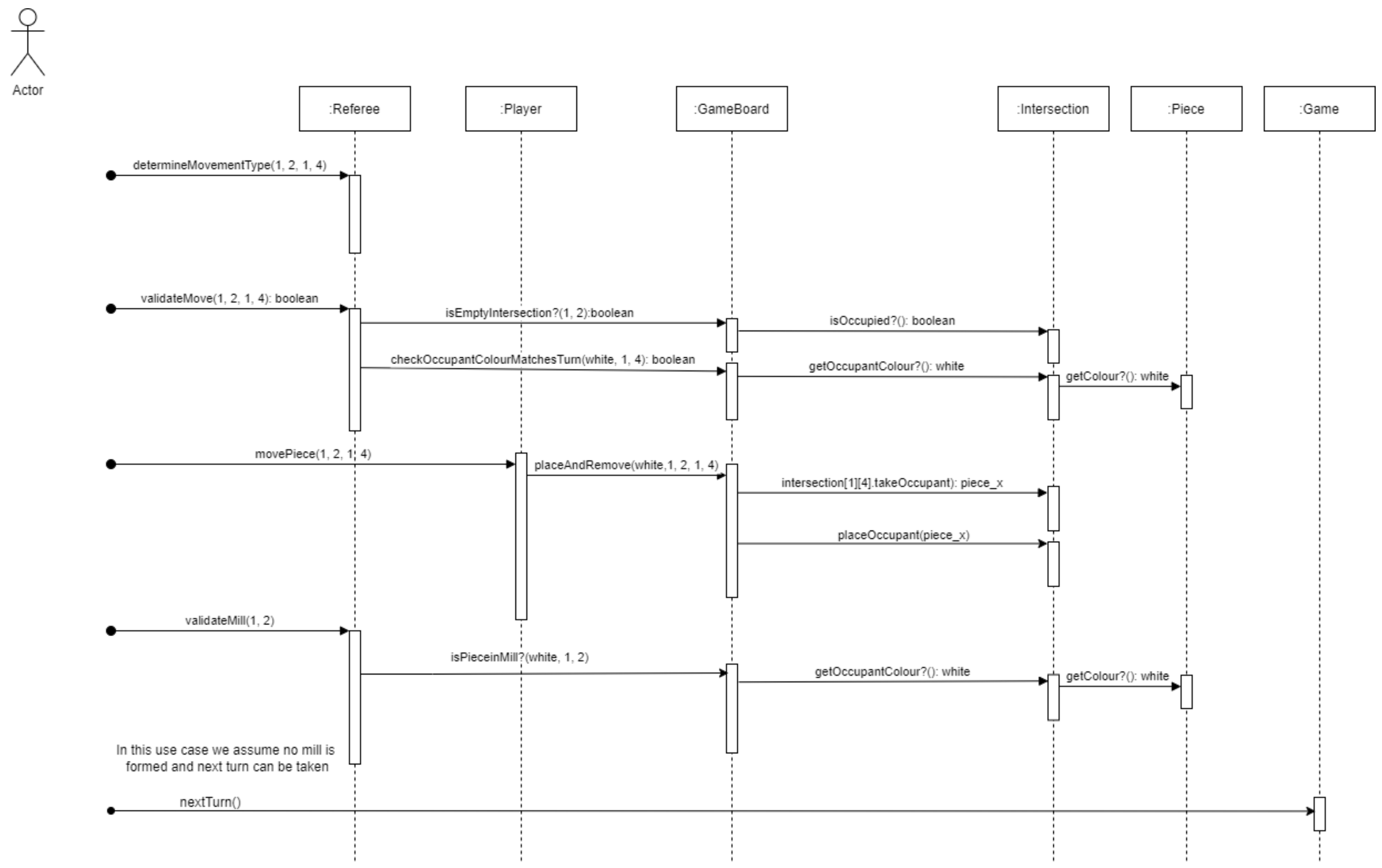

Referee
validatePlacement(x;int, y:int): Boolean
validateMove(x;int, y:int, piece_x;int, piece_y:int): Boolean
validateFly(x;int, y:int, piece_x;int, piece_y:int): Boolean
validateCapture(x;int, y:int): Boolean
validateMill(x:int, y:int): Boolean
determineMovementType(x:int, y:int, piece_x:int, piece_y:int): enum[:move, :fly]
checkWin(): Boolean
Supervises >
Supervises >
Game
setup()
nextTurn()
getTurnColour?(): enum[:white, :black]
getPhase?(): enum[:place, :move]
nextPhase()
endGame()
proposeDraw()
respondDraw(response: String)
forfeit()
getWinner()?: enum[:white, :black, :none]
0..*
Played by>
Player
placePieceOnBoard(x:int, y:int, piece_to_place: Piece)
movePiece(x:int, y:int, piece_x:int, piece_y:int)
capturePiece(x:int, y:int)
getColour?(): enum[:white, :black]
isBagEmpty?(): Boolean
1
<Plays game on
1
1
Has>
1
1
Controls >
Bag
selectPiece(): Piece
storePiece(piece:Piece)
isEmpty?(): Boolean
emptyBag()
1
Contains >
0..9
9
GameBoard
clearAllIntersections()
isEmptyIntersection?(x:int, y:int): Boolean
countPiecesOnBoard(colour:enum[:white, :black]):int
placeAnRemovePieces(x:int, y:int, piece_x:int, piece_y:int)
placePiece(x:int, y:int, piece:Piece)
removePiece(x:int, y:int)
checkOccupantColourMatchesTurn(turn_colour:enum[:white, :black, x:int, y:int): Boolean
isPieceInMill?(colour:enum[:white, :black], x:int, y:int): Boolean
existPieceNotInMill?(colour:enum[:white, :black]):Boolean
< Exists on
1
24
Intersection
takeOccupant(): Piece
isOccupied?(): Boolean
getXAsInteger?(): int
clear()
placeOccupant(piece:Piece)
getOccupantColour?(): enum[:white, :black]
Contains >
1
0..1
Piece
getColour?(): enum[:white, :black]

Preliminary Sequence Diagrams

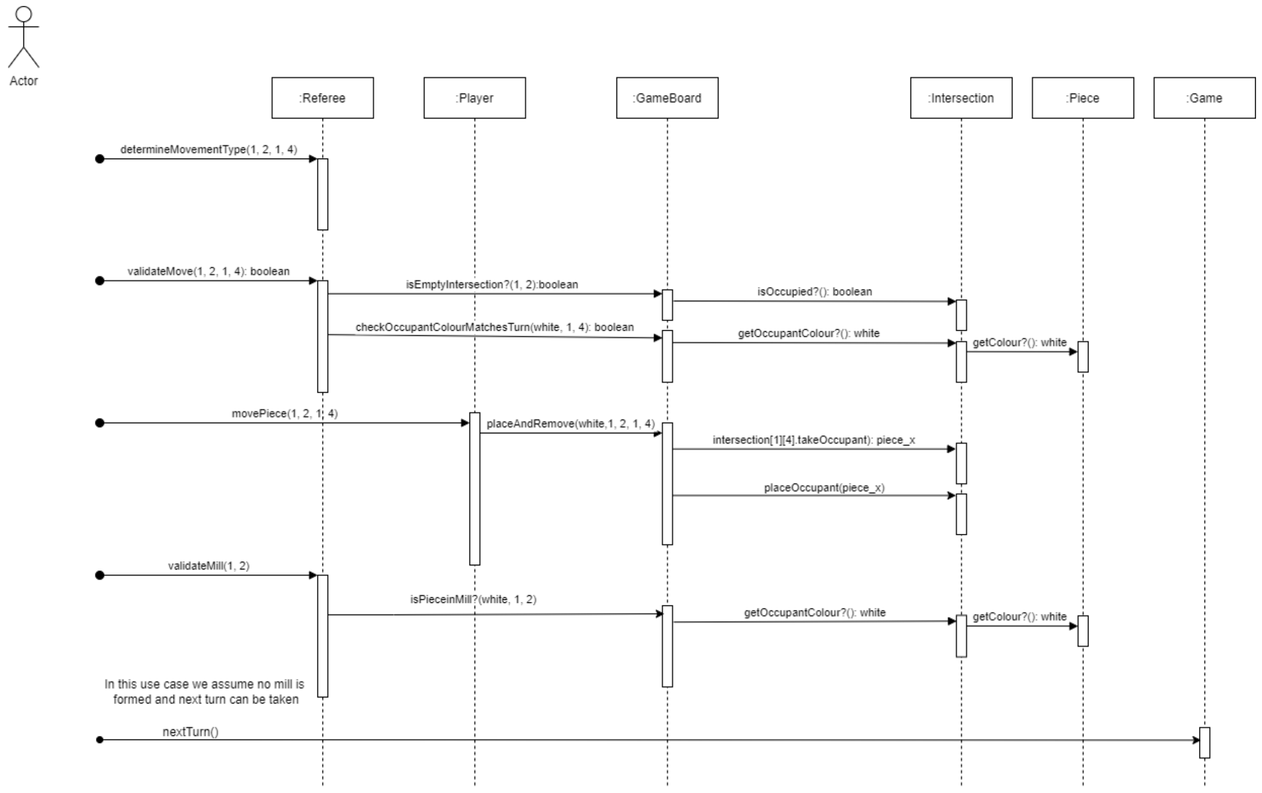
Use Case 1: Setup Game



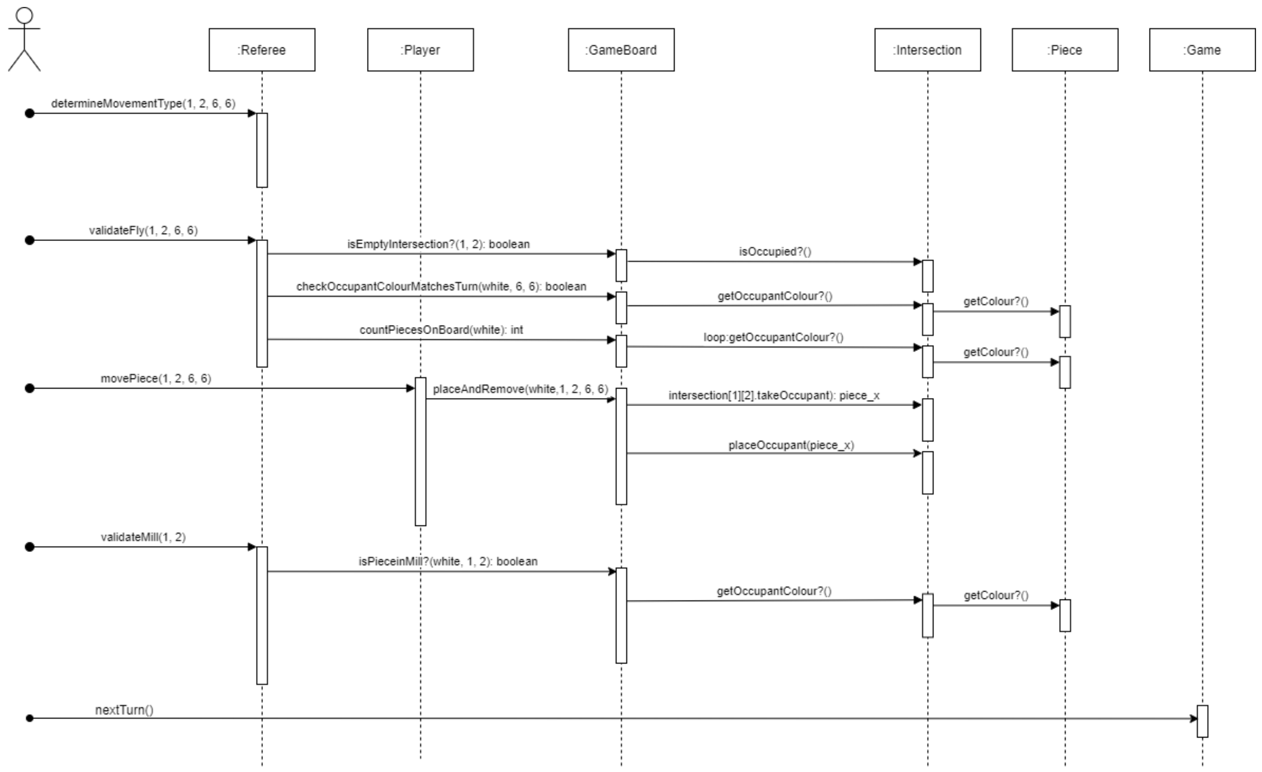
Use Case 2: Place game Piece


Actor
:Referee
:Player
:GameBoard
:Intersection
:Piece
:Game
determineMovementType(1, 2, 1, 4)
validateMove(1, 2, 1, 4): boolean
isEmptyIntersection?(1, 2):boolean
isOccupied?(): boolean
checkOccupantColourMatchesTurn(white, 1, 4): boolean
getOccupantColour?(): white
getColour?(): white
movePiece(1, 2, 1, 4)
placeAndRemove(white,1, 2, 1, 4)
intersection[1][4].takeOccupant): piece_x
placeOccupant(piece_x)
validateMill(1, 2)
isPieceinMill?(white, 1, 2)
getOccupantColour?(): white
getColour?(): white
In this use case we assume no mill is formed and next turn can be taken
nextTurn()

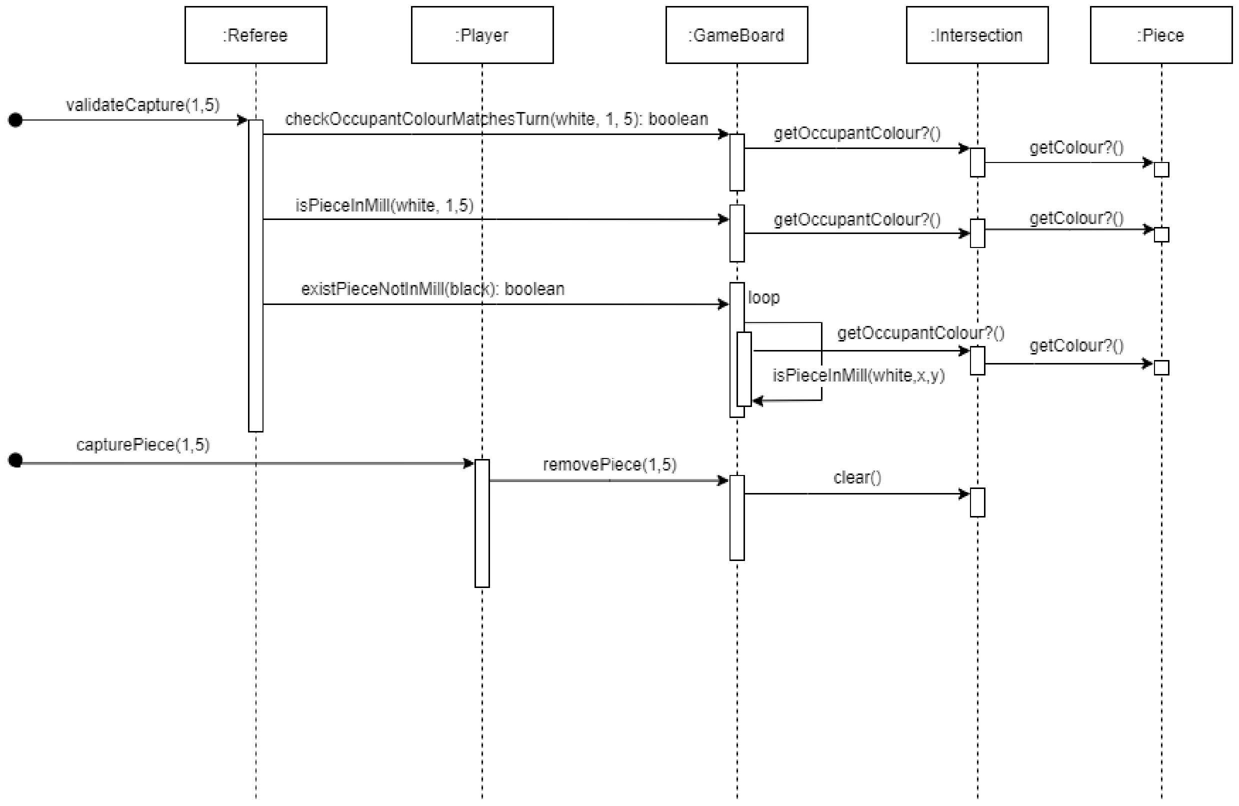
Use Case 3: Move game Piece


Actor
:Referee
:Player
:GameBoard
:Intersection
:Piece
:Game
determineMovementType(1, 2, 1, 4)
validateMove(1, 2, 1, 4): boolean
isEmptyIntersection?(1, 2):boolean
isOccupied?(): boolean
checkOccupantColourMatchesTurn(white, 1, 4): boolean
getOccupantColour?(): white
getColour?(): white
movePiece(1, 2, 1, 4)
placeAndRemove(white,1, 2, 1, 4)
intersection[1][4].takeOccupant): piece_x
placeOccupant(piece_x)
validateMill(1, 2)
isPieceinMill?(white, 1, 2)
getOccupantColour?(): white
getColour?(): white
In this use case we assume no mill is formed and next turn can be taken
nextTurn()

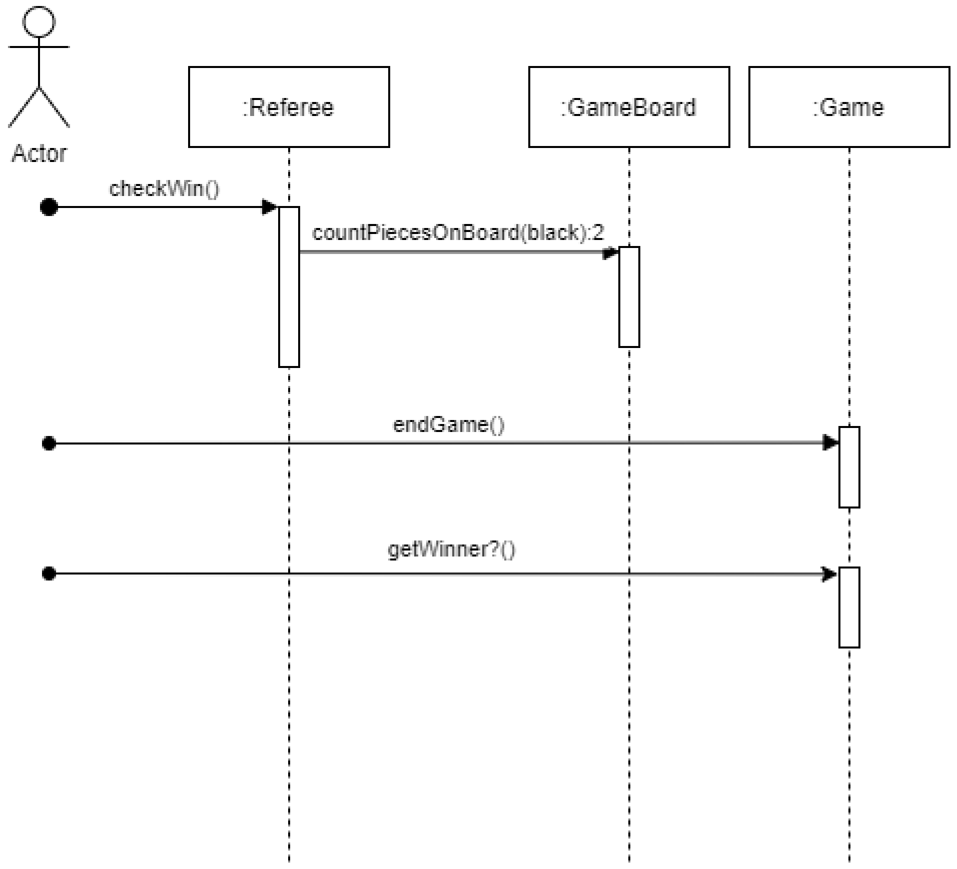
Use Case 4: Fly game Piece


:Referee
:Player
:GameBoard
:Intersection
:Piece
:Game
determineMovementType(1, 2, 6, 6)
validateFly(1, 2, 6, 6)
isEmptyIntersection?(1, 2): boolean
isOccupied?()
checkOccupantColourMatchesTurn(white, 6, 6): boolean
getOccupantColour?()
getColour?()
countPiecesOnBoard(white): int
loop:getOccupantColour?()
getColour?()
movePiece(1, 2, 6, 6)
placeAndRemove(white,1, 2, 6, 6)
intersection[1][2].takeOccupant): piece_x
placeOccupant(piece_x)
validateMill(1, 2)
isPieceinMill?(white, 1, 2): boolean
getOccupantColour?()
getColour?()
nextTurn()

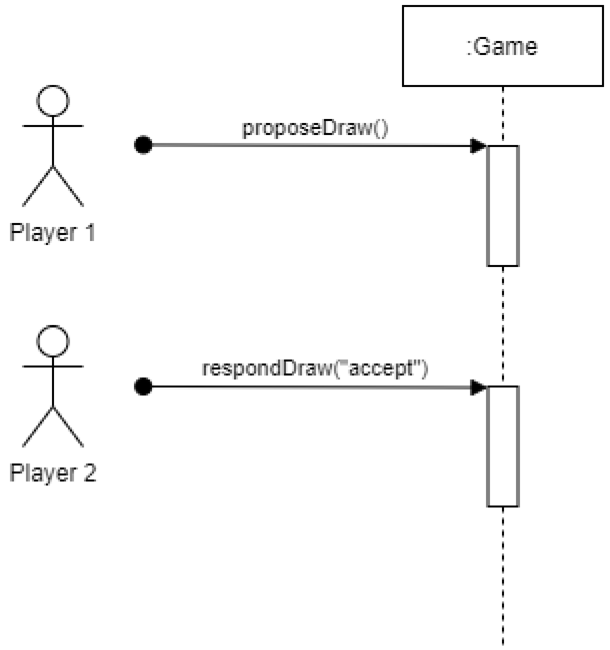
Use Case 5: Capture game Piece



Use Case 6: End the Game



Use Case 7: Propose a Draw



Remote Use Cases

In these use cases players send messages back and forth to each other regarding actions they are taking in the game. This is done through the use of a messenger service but simply described as "sends message". This is because the idea of sending a message is abstract and the message could be sent via a number of means depending on how this system is implemented. This is represented in the entity diagram as the 'Messenger' entity.

Remote Use Case 1

Use Case: Setup Game
Primary Actor: Player
Goal: To ensure players and the game board are ready for a game
Stakeholders List:
- Player
- Opponent
- Referee

Initiating Event:
Main Success Scenario:
1. Referee sends message to each player specifying whether they are white or black
2. Player clears and sets up board
3. Opponent clears and sets up board
4. Player sends message to opponent stating that they are ready to start the game
5. Opponent receives request to start game
6. Opponent sends message to Player stating that they are also ready to start the game

Post Conditions:
- Game has started
- White player is ready to make their first move

Alternate Flows or Exceptions: N/A

Use Cases Utilized: N/A
Scenario Notes:

Remote Use Case 2

Use Case: Place game piece
Primary Actor: Player

Goal: A player wants to place one of their off-board game pieces on an empty intersection on the board.
Stakeholders List:

- Player - Wants to place a game piece on the board
- Opponent
- Referee

Initiating Event: The game is in the first phase (placing phase) and it is the player's turn to place one of their pieces on the board.
Main Success Scenario:

1. Player selects one of their remaining pieces
2. Player sends message to Referee detailing where they want to play their piece
3. Referee determines that the placement is valid
4. Player places game piece on an empty intersection on the board
5. Player creates and sends message to opponent describing their move
6. Opponent receives message from the player
7. Opponent updates their board

Post Conditions:

- Player has one less game piece remaining off the board
- Game piece has been added to board

Alternate Flows or Exceptions:

1.a. Player runs out of pieces to place
   1.a.1) Referee determines that the first phase of the game is complete (placing phase)
   1.a.2) Referee sends message to both players informing them that the second phase of the game has started
2.a. Referee determines that placement is invalid
   2.a.1) Referee sends message to player notifying them the placement is invalid
   2.a.2) Player chooses a new intersection to place piece

Use Cases Utilized: N/A

Scenario Notes: Known as the placing phase. All 9 pieces must be placed by each player.

Remote Use Case 3

Use Case: Move game piece
Primary Actor: Player

Goal: The player wants to move one of their game pieces from its current intersection to an adjacent intersection.
Stakeholders List:
- Player - Wants to move piece to an adjacent intersection
- Opponent
- Referee

Initiating Event: All pieces have been added to the board and the player's turn has begun.
Main Success Scenario:
1. Player chooses a game piece that they want to move
2. Player sends message to Referee detailing where they want to move their piece
3. Referee determines that the move is valid
4. Player moves game piece to an adjacent intersection
5. Player creates a message detailing their move
6. Player sends message to opponent
7. Opponent receives message from the player
8. Opponent updates their board

Post Conditions:
- Game piece has been moved to an empty adjacent intersection
- The intersection that the game piece used to occupy is now empty

Alternate Flows or Exceptions:

3.a. Referee determines that movement is invalid

3.a.1) Referee sends message to player stating that movement is invalid

3.a.2) Player chooses a different action to take

Use Cases Utilized: N/A

Scenario Notes: The player has more than 3 game pieces remaining on the board, so they can only move pieces to adjacent, empty intersections.

Remote Use Case 4

Use Case: Fly game piece
Primary Actor: Player
Goal: The player wants to move one of their pieces to any non-adjacent, empty intersection on the board.
Stakeholders List:
- Player - Wants to move a game piece using a "Fly"
- Opponent

- Referee

Initiating Event: It's the player's turn and they only have 3 pieces remaining.
Main Success Scenario:
1. Player chooses a game piece that they want to move
2. Player sends message to Referee detailing where they want to move their piece
3. Referee determines that the move is valid
4. Player moves their game piece to an empty non-adjacent intersection
5. Player creates a message detailing their move
6. Player sends message to opponent
7. Opponent receives message
8. Opponent updates their board

Post Conditions:
- Game piece has been moved to an empty intersection
- The intersection that the game piece used to occupy is now empty

Alternate Flows or Exceptions:

1.a. Referee determines that the action is invalid

1.a.1) Referee sends message to player stating that their action is invalid

1.a.2) Player chooses a new action to take

Use Cases Utilized: N/A

Scenario Notes: N/A

Remote Use Case 5

Use Case: Capture a piece
Primary Actor: Player
Goal: The player wants to capture (remove) an enemy game piece.
Stakeholders List:
- Player - Wants to capture an enemy piece
- Opponent
- Referee

Initiating Event: Player has moved a piece or placed a piece resulting in them having 3 pieces in a row horizontally or vertically.
Main Success Scenario:
1. Referee determines that Player has formed a mill with their last move
2. Player sends message to Referee detailing what piece they want to capture
3. Referee determines that the capture is valid
4. Player captures an opposing game piece

5. Player creates message detailing their move
6. Player sends message to opponent
7. Opponent receives message
8. Opponent updates their board

Post Conditions:

- Opposing game piece has been removed
- The intersection that the game piece occupied is now empty

Alternate Flows or Exceptions:

1.a. Referee determines that the capture is invalid

1.a.1) Referee sends message to player stating that their capture is invalid

1.a.2) Player chooses a difference piece capture

Use Cases Utilized: N/A

Scenario Notes: N/A

Remote Use Case 6

Use Case: End the game
Primary Actor: Player
Goal: The referee determines if the move made by the player will end the game.
Stakeholders List:

- Player moving - Wants to win the game
- Opponent
- Referee - Determines whether the game has ended and who the victor is

Initiating Event: Player creates a mill and claims the opponent's 3rd last piece.
Main Success Scenario:

1. Player captures an Opponent piece
2. Referee verifies that the opponent only has two pieces left
3. Referee determines the winning player
4. Referee sends a message to both players that the game is over
5. Both Players reset their respective game boards

Post Conditions:

- The game is over
- Winner is decided
- Players are prompted to play again

Alternate Flows or Exceptions:

1.a Referee determines that the opponent still has three or more pieces remaining

1.a.1) Referee determines that game is not in an end state
1.a.2) Referee messages both players declaring that the game is not over

Use Cases Utilized:
Use Case 5

Scenario Notes: N/A

Remote Use Case 7

Use Case: Propose Game Draw
Primary Actor: Player, Opponent Player
Goal: The player taking their turn wishes to propose a draw to determine if the game should end.
Stakeholders List:

- Player moving - Wants the game to be concluded with a draw
- Opposing player - Must accept or decline proposal to determine game state
- Referee

Initiating Event: Both players have their pieces in a position in which they can infinitely prevent each other from forming a mill needed to win.
Main Success Scenario:

1. Player creates a message proposing a draw
2. Player sends message to opponent
3. Opponent receives message
4. Opponent sends message to player accepting the draw
5. Referee ends the game with no winner

Post Conditions:

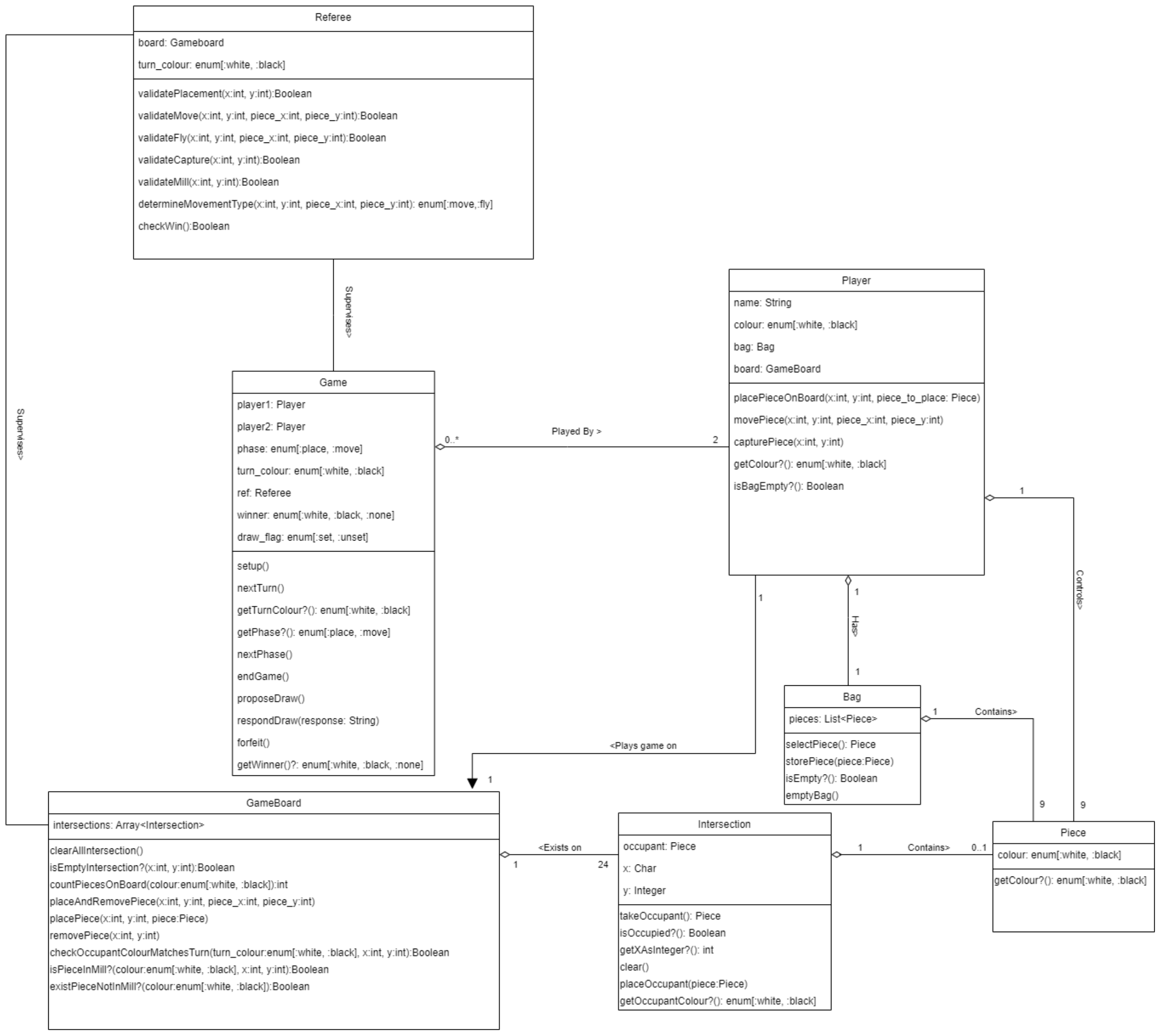
- The game is over
- No winner is decided
- Players are prompted to play again

Alternate Flows or Exceptions:
4.a. Opponent denies the draw proposal
4.a.1) Opponent reples with a message declining the draw

Use Cases Utilized: N/A
Scenario Notes: For the Draw case, it is difficult to determine a draw state, since both players can still move but may block each other from forming a mill infinitely. This case is impossible to implement algorithmically.

Detailed Design Components

Application - Class Diagram


Referee
board: Gameboard
turn_colour: enum[:white, :black]
validatePlacement(x:int, y:int):Boolean
validateMove(x:int, y:int, piece_x:int, piece_y:int):Boolean
validateFly(x:int, y:int, piece_x:int, piece_y:int):Boolean
validateCapture(x:int, y:int):Boolean
validateMill(x:int, y:int):Boolean
determineMovementType(x:int, y:int, piece_x:int, piece_y:int): enum[:move,:fly]
checkWin():Boolean
Supervises>
Supervises>
Game
player1: Player
player2: Player
phase: enum[:place, :move]
turn_colour: enum[:white, :black]
ref: Referee
winner: enum[:white, :black, :none]
draw_flag: enum[:set, :unset]
setup()
nextTurn()
getTurnColour?(): enum[:white, :black]
getPhase?(): enum[:place, :move]
nextPhase()
endGame()
proposeDraw()
respondDraw(response: String)
forfeit()
getWinner()?: enum[:white, :black, :none]
0..*
Played By >
2
Player
name: String
colour: enum[:white, :black]
bag: Bag
board: GameBoard
placePieceOnBoard(x:int, y:int, piece_to_place: Piece)
movePiece(x:int, y:int, piece_x:int, piece_y:int)
capturePiece(x:int, y:int)
getColour?(): enum[:white, :black]
isBagEmpty?(): Boolean
1
1
Has>
1
1
Controls>
<Plays game on
1
Bag
pieces: List<Piece>
selectPiece(): Piece
storePiece(piece:Piece)
isEmpty?(): Boolean
emptyBag()
1
Contains>
9
9
GameBoard
intersections: Array<Intersection>
clearAllIntersection()
isEmptyIntersection?(x:int, y:int):Boolean
countPiecesOnBoard(colour:enum[:white, :black]):int
placeAndRemovePiece(x:int, y:int, piece_x:int, piece_y:int)
placePiece(x:int, y:int, piece:Piece)
removePiece(x:int, y:int)
checkOccupantColourMatchesTurn(turn_colour:enum[:white, :black], x:int, y:int):Boolean
isPieceInMill?(colour:enum[:white, :black], x:int, y:int):Boolean
existPieceNotInMill?(colour:enum[:white, :black]):Boolean
<Exists on
1
24
Intersection
occupant: Piece
x: Char
y: Integer
takeOccupant(): Piece
isOccupied?(): Boolean
getXAsInteger?(): int
clear()
placeOccupant(piece:Piece)
getOccupantColour?(): enum[:white, :black]
1
Contains>
0..1
Piece
colour: enum[:white, :black]
getColour?(): enum[:white, :black]

Application - Detailed Class List

Important Information Regarding Method Parameters

In this game design, the web end of the design presents a game board that makes use of a letter and number based coordinate system so that users understand which select box is for each game axis without having to know what an X or Y axis is. The application however, uses an array for gameboard intersection storage. This means that all x and y values that are passed to the application subsystem should be converted to integer form prior to use. For example, if the user selects the coordinate ‘a, 7’, the character value should be converted to a 0-6 scale for use in the Gameboard intersections array. The resulting value of this conversion would be ‘0,6’.

Detailed Processes and Pseudocode

For many functions that are slightly more complex we have provided a detailed process and/or pseudocode to help with implementation. If the pseudocode is rather lengthy we have also provided curly braces within the pseudocode to assist in making it easier to read. The pseudocode should not simply be copied as it is not guaranteed to be accurate and syntax will not match that of real Ruby code.

Referee

Instance Variables

- board: GameBoard
- turn_colour: enum[:white, :black]

Constructors

- ***initialize(board: GameBoard, turn_colour: enum[:white,:black])***
    - Creates a Referee object containing the sole gameboard of the game (explained in controller class details) and the colour of the player currently taking their turn.

Methods

- ***validatePlacement(x:int, y:int): Boolean***
    - Check if the intersection where a player wants to place their piece is unoccupied

      Detailed Process

      1) Using the provided x and y coordinates, check the referee’s Board Intersections array to see if the location is currently occupied
      2) If the intersection is not occupied, the placement is valid, return true
      3) If the intersection is occupied, the placement is not valid, return false

      Pseudocode

      ```
      if(board.isEmptyIntersection?(x, y) == false) {
          Return false
      }
      Return true
      ```

- ***validateMove(x:int, y:int, piece_x:int, piece_y:int): Boolean***
    - Check if the piece the player wants to move belongs to them and that the intersection where a player wants to move their piece to is unoccupied

      Detailed Process
      1) Using the provided x and y coordinates, check the referee's Board Intersections array to see if the location is currently occupied
          a) If the intersection is occupied, the movement is not valid, return false
      2) Check to see if the occupant colour of the intersection that the player wishes to move the piece from (piece_x, piece_y) matches their colour
          a) If the occupant of the intersection does not match the turn_colour, the movement is not valid, return false
      3) If both checks are true, then the move is valid, return true

      Pseudocode
      ```
      // check if destination is empty
      if( board.isEmptyIntersection?(x, y) == false)
              Return false

      // check if piece matches colour of the player taking turn
      If (checkOccupantColourMatchesTurn(self.turn_colour, piece_x, piece_y)
       == false)
              Return false

      Return true
      ```

- ***validateFly(x:int, y:int, piece_x:int, piece_y:int): Boolean***
    - Check if the player is allowed to execute a 'fly' move (only has three pieces remaining on the board), that the piece they want to move belongs to them and if the intersection where a player wants to move their piece to is unoccupied

      Detailed Process
      1) Using the provided x and y coordinates, check the referees Board Intersections array to see if the location is currently occupied
          a) If the intersection is occupied, the movement is not valid, return false
      2) Check to see if the occupant colour of the intersection that the player wishes to move the piece from (piece_x, piece_y)
          a) If the occupant of the intersection does not match the turn_colour, the movement is not valid, return false
      3) Check that the player only has 3 pieces on the board (fly condition)
          a) If not, the move is not valid, return false

4) If all checks are true, then the move is valid, return true

Pseudocode

```
// check if destination is empty
if( board.isEmptyIntersection?(x, y) == false){
        Return false
}

// check if piece matches colour of the player taking turn
If (board.checkOccupantColourMatchesTurn(turn_colour, piece_x, piece_y) ==
false) {
        Return false
}

// check that the player only has 3 pieces on the board
If ( board.CountPiecesOnBoard(turn_colour) != 3) {
        Return false
}
Return true
```

- ***validateCapture(x:int, y:int): Boolean***
    - Check if the piece the player wants to capture is the correct colour (the opposite of turn_colour), and either not part of a mill or is part of a mill and there are no remaining opponent pieces that aren't part of a mill.

      Detailed Process

      1) Check if piece at gameboard intersections is of the opposing colour
          a) If so check that if piece is part of a mill
              i) If true ask the gameboard if there are any opposing pieces that aren't in a mill
                  (1) If there aren't then the capture is valid, return true
                  (2) If there is then the capture is not valid, return false

      Pseudocode

```
if (board.checkOccupantColourMatchesTurn != turn_colour)
        if (board.isPieceInMill(opposite_of_turn_colour, x, y) == true)
                //check if there are any opponent pieces on the board that are not
                part of a mill
                if (existPieceNotinMill?(opposite_of_turn_colour) == false)
                        Return true
                else
                        Return false
        else
                Return true
```

```
else
        return false
```

- ***validateMill(x:int, y:int): Boolean***
    - Check if the player has formed a mill (3 pieces of the player's colour either vertically or horizontally in a line)

      Detailed Process
      1) Checks with gameboard to see if piece at intersections[x][y] is part of a mill
          a) Returns true if the piece at the intersection is part of a mill
          b) False if not part of a mill

      Pseudocode
      ```
      if (board.isPieceInMill(turn_colour, x, y) == true)
              Return true
      else
              Return false
      ```

- ***determineMovementType(x:int, y:int, piece_x:int, piece_y:int): enum[:move,:fly]***
    - Determines if the movement the player is attempting to do is a "fly" or a regular "move". If the player is attempting to move their piece to an adjacent intersection, then it's a regular "move". If the player is attempting to move to a non-adjacent intersection then it is a "fly". This information is used to determine which validation method needs to be called: ValidateMove() or ValidateFly().

      Parameters
      - x,y: the coordinates to move the piece to, in the gameboard's intersections instance variable
      - piece_x, piece_y: the coordinates to get the piece from, in the gameboard's intersections instance variable

      Detailed Process
      - Optional Error Checking: check that the coordinate values exist on the gameboard
      1) Check if the two provided sets of coordinates are adjacent to each other. This could be adjacency in the x or y direction (ex. horizontal or vertical), we do not check diagonal adjacency as pieces cannot move on the diagonal, except for the condition of a fly.
          a) If the coordinates are adjacent, the move type is a standard move
              i) return :move
          b) If the coordinates are not adjacent, the move type is a fly
              i) return :fly

Pseudocode - Algorithm for Adjacency
This is an example of a hardcoded solution that could be used to check adjacency. There are likely better mathematical implementations that could be used. Please refer to the board constructor for details on how the intersections are located in the application.

```
switch(piece_x,piece_y):
    case(0, 0):
        If (x == 0 && y == 3 || x == 3 && y == 0)
            Return true
        Else
            Return false
```

- ***checkWin(): Boolean***
    - Check if the opposing colour player has less than 3 pieces left on the board. If the opponent has less than 3 pieces left, then they have lost. If this is the case, then return true as the player taking their turn has currently won.

      Detailed Process
      1) Check if the opposing colour/player (if turn_colour is :white then we are checking :black, and vice versa) has less than 3 pieces left on the board.
          a) If that colour does has less than 3 pieces left than the player currently taking their turn wins

      Pseudocode

```
if turn_colour == :white
    if board.countPiecesOnBoard(:black) < 3
        return true //white has won, black has lost
    else
        return false //nobody has won yet
if turn_colour == :black
    if board.countPiecesOnBoard(:white) < 3
        return true //black has won, white has lost
    else
        return false //nobody has won yet
```

Game

Instance Variables

- player1: Player
- player2: Player
- phase: enum[:place, :move]
- turn_colour: enum[:white, :black]
- ref: Referee
- winner: enum[:white, :black, :none]
- draw_flag: enum[:set, :unset]

Constructors

- ***initialize(player1: Player, player2: Player, phase: enum[:place, :move], turn_colour: enum[:white, :black])***
    - Creates a game object that possesses both players, the current game phase, the turn colour of the turn being taken, and a referee

Methods

- ***setup()***
    - This function is used to set up all of the initial states required for a game to be played. This should set the turn_colour to *:white* and the phase to *place*, as well as set the colours of each player.

      Detailed Process
        1) Set turn_colour to :white (white goes first)
        2) Set the colour of each player to either white or black(this should be randomized)
        3) Set game phase to place as a new game is beginning

- ***nextTurn()***
    - Update the Game's state so that it is now the other player's turn and notify each player of who's turn it is.

      Detailed Process
        1) Determine which colour player just made their move
        2) If still in the place phase, check to see if both players have placed all the pieces from their bag
            a) If so shift to the 'move' phase

      Pseudocode

```
if (turn_colour == :white)
        turn_colour = :black
else if (turn_colour == :black)
        turn_colour = :white
if (phase == place)
        Call IsBagEmpty? On each player
```

If both player’s return true, then call NextPhase() (The place phase ends once each player has placed all their pieces, it then moves onto the move phase)

- ***getTurnColour?(): enum[:white,:black]***
    - Returns the symbol of the current turn colour (the instance variable turn_colour)

- ***getPhase?(): enum[:place,:move]***
    - Returns the symbol of the current phase that the game is in (the instance variable phase).

- ***nextPhase()***
    - Transition the game from the “place” phase to the “move” phase
    - This phase variable is used to determine what the user is able to do on their turn
    - Set phase to ‘move’

- ***endGame()***
    - This function is used to declare the winner of the game to be the player colour whose turn it currently is.

      Detailed Process
        1) Set the winner instance variable to be what turn_colour currently is
            a) Ex: if turn_colour is :white then winner should be set to :white

- ***proposeDraw()***
    - This function sets the draw flag within the game
    - draw_flag instance variable should be set to ‘set’

- ***respondDraw(response: String)***
    - This function is used to determine if the game is ending based on the response to a draw. If the draw flag is set, and the draw was accepted, the winner should be set to ‘none’.

      Detailed Process
        1) Check if the draw flag is set
            a) If not raise an error and return
        2) Check if response is ‘accept’
            a) If so set instance variable winner to be ‘none’ and return
        3) If response is ‘decline’
            a) set draw_flag to ‘unset’ and return

- ***forfeit()***
    - This function is used to declare the winner of the game to be the player whose turn it is currently not (a player may only forfeit on their own turn).

Detailed Process

1) If the current turn_colour is :black, then set winner to be :white
2) If the current turn_colour is :white, then set the winner to be :black

- ***getWinner?(): enum[:white,:black,:none]***
  - This function returns the value of the winner instance variable from the game

Player

Instance Variables

- name: String
- colour: enum[:white, :black]
- bag: Bag
- board: GameBoard

Constructors

- ***initialize(Name: String, Colour: enum[:white, :black])***

Methods

- ***placePieceOnBoard(x:int, y:int, piece_to_place: Piece)***
  - Place one of the player's pieces on the board at intersection x,y

    Detailed Process

    1) Place the piece on the gameboard at intersection x,y

    Pseudocode

    ```
    board.placePiece(x, y, piece_to_place)
    ```

- ***movePiece(x:int, y:int, piece_x:int, piece_y:int)***
  - Move a piece from intersection Piece_x, Piece_y to intersection x,y. Upon success, the intersection the player moved from should be vacant and the intersection the player moved to should be occupied by the piece they moved.

    Parameters

    - x,y: the intersection coordinates that the player is moving a piece to
    - piece_x, piece_y: the intersection coordinates that a player is moving a piece from

    Pseudocode

    ```
    board.placeAndRemovePiece(x,y,piece_x,piece_y)
    ```

- ***capturePiece(x:int, y:int)***
  - Capture the piece at intersection x,y. Upon success, the piece is removed from the board and cannot be replayed. The intersection where it was is now vacant/empty.

Pseudocode

```
board.removePiece(x,y)
```

- ***getColour?(): enum[:white, :black]***
    - Returns the colour instance variable of the player

- ***isBagEmpty?(): Boolean***
    - Call IsEmpty?() on the player's bag and return true or false

Bag

Instance Variables

- pieces: List<Piece>

Constructors

- ***initialize()***
    - *pieces = [ ] on initialization (empty list/array)*

Methods

- ***selectPiece(): Piece***
    - Removes a piece from the bag's list of pieces and returns the object (the piece should no longer exist in the bag)
    - If no remaining pieces in bag, return nil

- ***storePiece(piece:Piece)***
    - Append a new piece to the Bag's list of pieces, the location where it is appended is irrelevant

- ***isEmpty?(): Boolean***
    - Checks to see if the bags list of pieces contains no more objects
        - Essentially checking if length of the list is 0
    - If the Bag's list of pieces is empty, return true
    - Else return false

- ***emptyBag()***
    - Set the Bag's list of pieces to nil.

Piece

Instance Variables

- *colour: enum[:white, :black]*

Constructors

- ***initialize(colour:enum[:white, :black])***
    - Creates a new piece of colour :white or :black

  - Colour will be dependent on what player the piece is being created for (if the player is the :white or :black player)

Methods

- ***getColour?(): enum[:white, :black]***
  - Returns the colour instance variable of the piece

GameBoard

Instance Variables

- intersections: Array<Intersection>
  - The intersection objects that are present on the game board are stored in this array.

Due to the complex nature of the gameboard object, a 2D array storing the pieces was used for easiest possible computation of mills. Below is a table that demonstrates the layout of the array where cells that have ‘x,y’, would possess an intersection object from that intersection on the board, and cells that have the term “null” in them, would not possess an object. This is clarified with pseudocode to create this array structure (see the constructor details).

Note: The array index values are presented in the leftmost column and bottom row.

6	*a,7*	*null*	*null*	*d,7*	*null*	*null*	*g,7*
5	*null*	*b,6*	*null*	*d,6*	*null*	*f,6*	*null*
4	*null*	*null*	*c,5*	*d,5*	*e,5*	*null*	*null*
3	*a,4*	*b,4*	*c,4*	*null*	*e,4*	*f,4*	*g,4*
2	*null*	*null*	*c,3*	*d,3*	*e,3*	*null*	*null*
1	*null*	*b,2*	*null*	*d,2*	*null*	*f,2*	*null*
0	*a,1*	*null*	*null*	*d,1*	*null*	*null*	*g,1*
	0	***1***	***2***	***3***	***4***	***5***	***6***

Constructors

- ***initialize(intersectionsForBoard<Array>: intersections)***

  Constructor Process Details
  Due to the intersections x value being a character this board will have to do some processing to determine where to place each intersection in the array.
  1) For each intersection in the intersectionsForBoard parameter do the following
     a) Get the numeric value of the x coordinate of the intersection

   b) Get the numeric value of the y coordinate of the intersection (stored as integer by default)
   c) Subtract 1 from the Y as our board array needs to start at 0
      i) We don't need to do this for X as it is precalculated on the character conversion (see method details for *getXAsInteger*)
   d) Save the intersection to the boards array of intersection at these respective x and y integer values

Pseudocode

```
for each item in intersectionsForBoard
        self.intersections[item.getXAsInteger?][ (item.getY? - 1)] = item
```

Methods

- ***clearAllIntersections()***
    - This function removes the occupants from each intersection on the gameboard.

      Detailed Process
      1) Loop through every intersection on the board
      2) For each Intersection, call clear() method

- ***isEmptyIntersection?(x:int, y:int): Boolean***
    - This function checks to see if a specified intersection at x,y currently has an occupant. If no occupant is present this function should return false.

      Detailed Process
      1) Check and see if an intersection exists at intersections[x][y]
         a) If this value is nil, then return false, this will mark the move as invalid
      2) Check if the intersection at x,y has an occupant
         a) If it doesn't, return true. If it does, return false

      Pseudocode

```
If (self.intersections[x][y] == nil)
        return false
If (self.intersections[x][y].isOccupied? == false)
        return true
else
        return false
```

- ***countPiecesOnBoard(colour:enum[:white, :black]): int***
    - Counts the number of pieces on the board of a certain colour and returns the number of pieces it counted.

Detailed Process / Pseudocode

```
1) Loop through all intersection in the gameboard's intersections array
2) For each intersection, call GetOccupantColour?()
    a) If the returned colour == colour param then increment count++
3) Return count
```

- ***placeAndRemovePiece(x:int, y:int, piece_x:int, piece_y:int)***
    - This function removes an occupant from an intersection on the board and places it on a new specified intersection. This is used when a player wants to move a piece via either a move or fly.

      Parameters
        - x,y: the coordinates to move the piece to, in the gameboard's intersections instance variable
        - piece_x, piece_y: the coordinates to get the piece from, in the gameboard's intersections instance variable

      Detailed Process
        1) Get the piece present at intersection with coordinates piece_x, piece_y
        2) Place this piece at intersection with coordinates x, y

      Pseudocode
      ```
      intersections[x][y].placeOccupant(intersections[piece_x][piece_y].takeOccupant?)
      ```

- ***placePiece(x:int, y:int, piece: Piece)***
    - Set the occupant of intersection at x, y to be the piece provided to the method. This is used by a player placing a piece on the gameboard during the placement phase.

      Pseudocode
      ```
      intersections[x][y].placeOccupant(piece)
      ```

- ***removePiece(x:int, y:int)***
    - Removes ('capture') a piece at intersection x,y
        - This is done by setting the occupant value to nil
    - Captured pieces are simply 'dropped' and collected by the garbage collector (no need to track them as they cannot return to play in the game)

- ***checkOccupantColourMatchesTurn(turn_colour:enum[:white,:black], x:int, y:int): Boolean***
    - This function checks to see if the piece on the intersection (x,y) that the player wishes to move from is actually the player's piece. If the piece is not the player's piece (it is a different colour than turn_colour) then return false.

Detailed Process

1) Check if the piece occupying intersection[x][y] matches the colour of the player whose turn it currently is
2) Call getOccupantColour?() on intersection[x][y]
3) If returned colour == param turn_colour then return true
    a) If colour is not the same, the move is invalid as a player cannot move a piece of another colour

Pseudocode

```
if (self.intersections[x][y].getOccupantColour? == turn_colour)
        Return true    // the occupant is the players piece
else
        Return false
```

- ***isPieceInMill?(colour:enum[:white,:black], x:int, y;int): Boolean***
    - Checks to see if the piece at a given coordinate is part of a valid mill

      Detailed Process

      1) Check each possible vertical mill for the piece location
          a) See if all pieces within the same vertical row of 3 of the piece are of the same colour
      2) Check each possible horizontal mill for the piece location
          a) See if all pieces within the same horizontal row of 3 of the piece are of the same colour
      3) If the piece was in either a vertical or horizontal mill it is confirmed to be in a mill and the function should return true
          a) If not in either type of mill return false

      Pseudocode (HIGHLY RECOMMENDED TO VIEW APPENDIX)

      - See **Appendix A** for a full algorithm implementation. We chose to place this in the appendix due to the length and complexity of the code.

- ***existPieceNotInMill?(colour:enum[:white,:black]): Boolean***
    - This function will check every intersection on the gameboard to see if there exists a piece of the specified colour that is not currently in a mill.

      Detailed Process

      1) Iterate through the list of intersections
      2) For each intersection, check if the occupant colour matches the colour param
          a) If True, check if the piece is in a mill
              i) If the piece is not in a mill, return true

3) At end of iteration and we have not found a single piece that is not in a mill, we want to return false, as there does not exist a piece not in a mill

Pseudocode

```
for each item in intersections:   // note that this would require nested for loops
        if (item != nil)
                // check the occupant colour
                if (item.getOccupantColour == colour)
                        // check to see if it is in a mill
                        // x and y would be for loop counters
                        if (isPieceInMill (colour, x, y) != true)
                                Return true
Return false
```

Intersection

Instance Variables

- occupant: Piece
- x: Char
- y: Integer

Constructors

- ***initialize(x: Char, y: Int)***
- ***initialize(x: Char, y: Int, occupant: Piece)***

Methods

- ***takeOccupant(): Piece***
    - Remove and return the occupant of the intersection (Occupant variable should be nil after this process)
    - If no occupant, return nil

- ***isOccupied?(): Boolean***
    - If occument variable is set, return true
    - Else return false

- **getXAsInteger?(): int**
    - Returns the integer value of the character
        - Note that we should subtract 97 for this system, as we want the ascii value of ‘a’ to be our base point of 0. This means that converting instance variable x with the coordinate ‘a’ will result in an integer of 0, ‘b’ would result in 1, so on and so forth.

Pseudocode

```
Return (x.ord - 97)
```

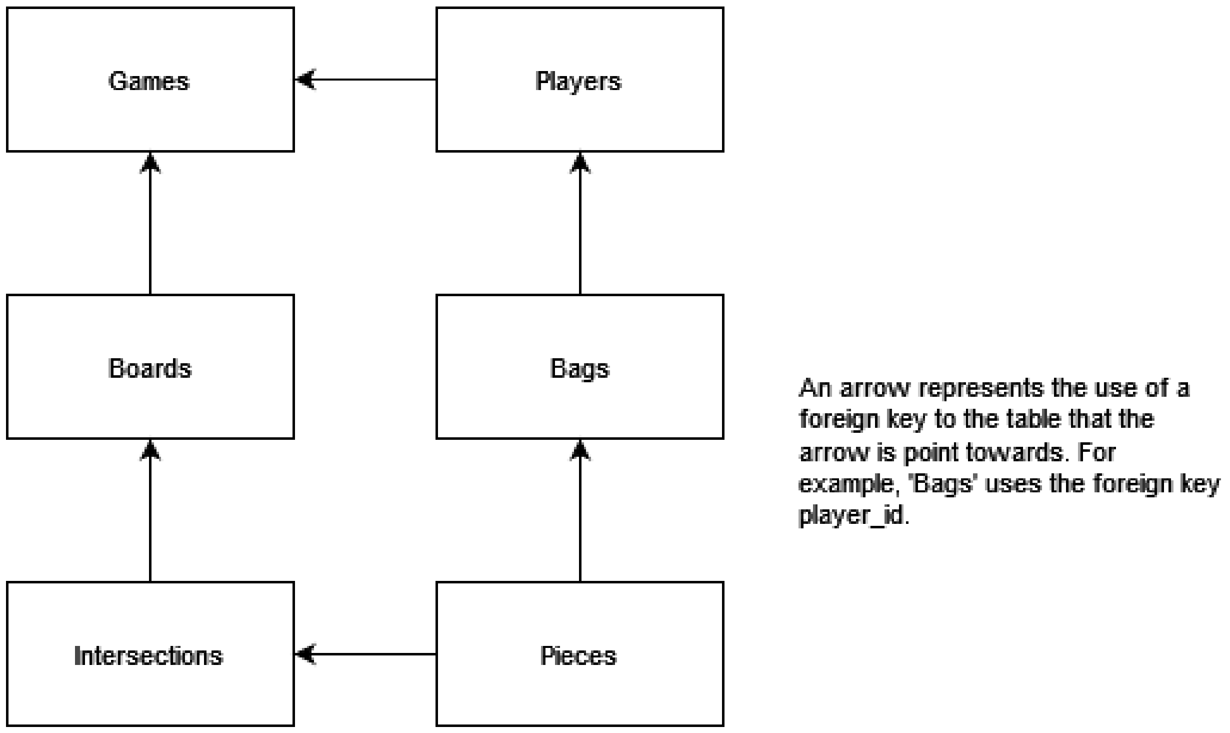
- ***clear()***
    - This function sets occupant instance variable to be nil

- ***placeOccupant(piece: Piece)***
    - This function should set the occupant instance variable to be the piece parameter

- **getOccupantColour?(): enum[:white, :black]**
    - Get the colour of an occupant that exists on an intersection. The colour should be returned as the symbol :white or :black.
    - Error checking: check to see if the intersection has an occupant before attempting to get the colour of the occupant. If no occupant is present, return nil.

      Pseudocode
      ```
      if (occupant != nil)
          return occupant.getColour?
      else
          return nil
      ```

Rails - Data Model


Games
Players
Boards
Bags
Intersections
Pieces
An arrow represents the use of a foreign key to the table that the arrow is point towards. For example, 'Bags' uses the foreign key player_id.

Simple Description: An arrow represents a potential join via foreign key in the above diagram. See figure text for example.

Table: Games

Field	**Type**
Primary Key - game_id	integer (key)
game_phase	enum[place, move]
game_status	enum[playing, complete]
winner	integer (copy of winning player_id), -1 if draw
draw_flag	enum[set, unset]

Table: Boards

Field	**Type**
Primary Key - board_id	integer (key)
Foreign Key[Games] - game_id	integer (key)

Table: Players

Field	Type
Primary Key - player_id	integer (key)
email	String
password	String (encrypted)
colour	enum[white, black]
wins	Integer (default: 0)
losses	Integer (default: 0)
Foreign Key[Games] - game_id	integer (key)

Table: Bags

Field	Type
Primary Key - bag_id	integer (key)
piece_list	[integer, integer,...] 0..9
Foreign Key[Players] - player_id	integer (key)

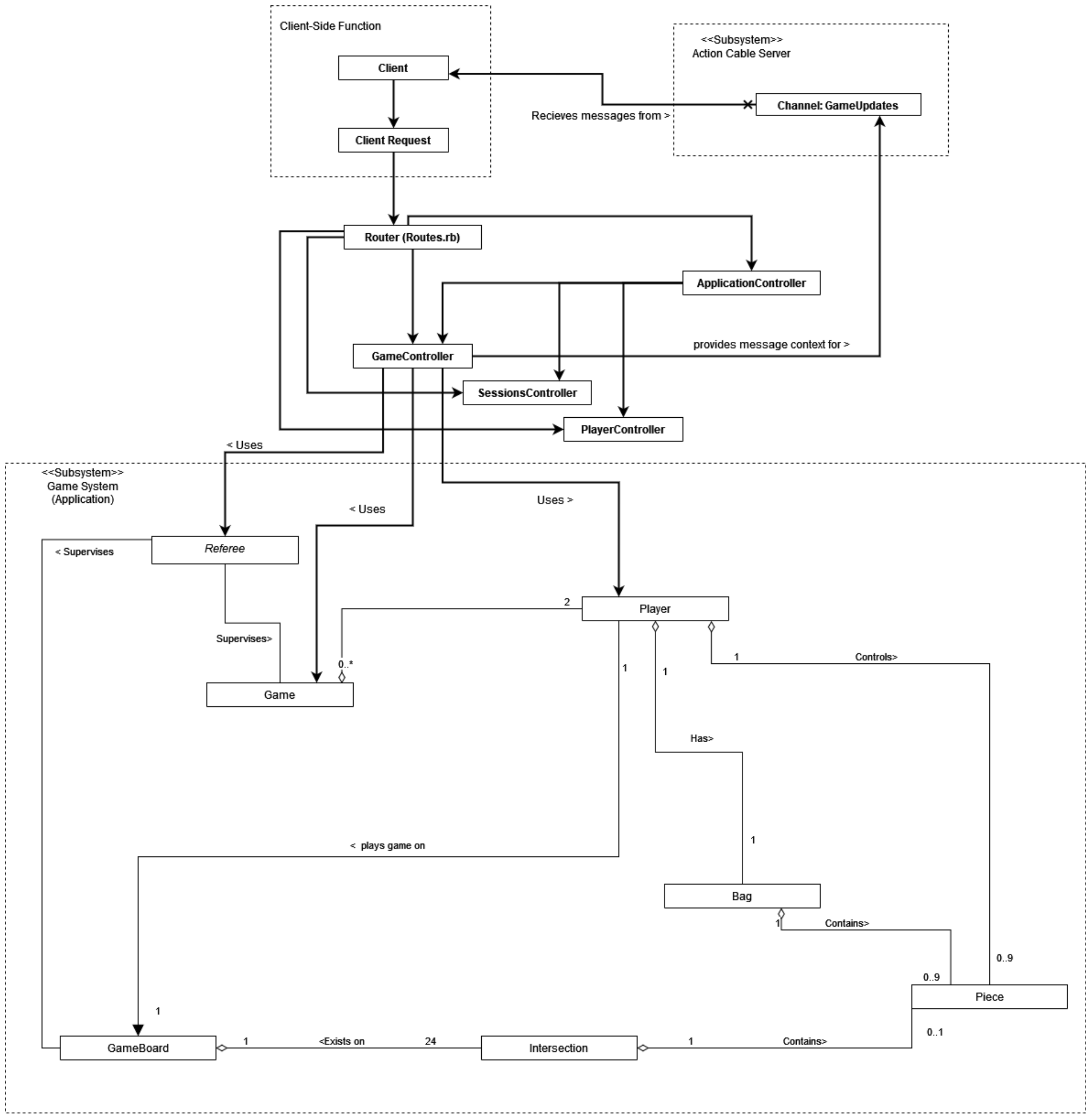
Table: Pieces (contained in a bag, or placed on a board)

Field	Type
Primary Key - piece_id	integer (key)
piece_status	enum[claimed, unclaimed]
colour	enum[white, black]
Foreign Key[Intersections] - intersection_id	integer (key)
Foreign Key[Bags] - bag_id	integer (key)

Table: Intersections

Field	**Type**
Primary Key - intersection_id	integer (key)
row	enum [1, 2, 3, 4, 5, 6, 7]
column	enum [a, b, c, d, e, f, g]
Foreign Key[Boards] - board_id	integer (key)

Rails - Controller Class Diagram



Rails - Controller Class Details

System Implementation Notes

When implementing this system design we recommend that a login system also be created for easier online play capabilities. When players first enter the website they will be met with a login/signup page and an account containing an email and a password should be created in order for them to start playing a game of *Nine Men's Morris*.

The controller class processes and descriptions retain a focus on how the controllers themselves operate with limited insight into the application classes. Please refer to the Class details list for information on how the Game operates. This being said, for almost all operations we provided a complete overview of steps that should be taken within the controller methods (referee validation as well as object creation). Some references to when the database tables should be used were also made to help simplify the coding of complex methods. However, it is safe to assume that whenever a stored class object is changed, it's values in the database should be updated as well.

Note: The commonly used term **message** refers to a message being sent from the GameController through the action cable channel: **GameUpdates** to the client(s).

ApplicationController.rb

Methods

- **authenticate_player()** - Before action is taken, check if the player has been authenticated (their session exists both locally and in the database). If they have not been authenticated, redirect them to the login page.

- **current_player()** - Use the player's session ID to look up the player in the database. If the player is not found then display an error message and clear the user's session.

SessionsController.rb

Methods

- ***check_session()*** - Check to see if a session for the user visiting the site exists, and if so automatically navigate them to the home player page. If not, redirect to the login page.

- ***create_session()*** - On login attempt check to see if the user exists and if they do navigate them to the home player page. This is the function where we also error check login information and provide error requests for bad email or password attempts.

  Detailed Process

  a. The system checks the database for the provided email and password

b. If a match is found, the system creates a session and redirects the user to the home page
c. If no match is found, the system sends a message to the user that their email/password is incorrect

PlayerController.rb

Methods

- ***create_account()*** - Create a new account using an email and password provided by the user. Alert the user if the email they used is already in use or if there are missing fields.

  Detailed Process

  a. Email and password are received
     - If the player already exists in the Players table, an alert is displayed telling the user
     - If the user missed a field, an alert is displayed telling the user what they must fill out in the web form

  b. On successful account creation
     - A new row (player) with the player information will be added to the Players table
     - A new bag will be added to the Bags table and linked to the new player via the player_id
     - The user will be redirected to the login page and provided a message saying “Account successfully created”

- ***logout_player()*** - Log the user out and delete their current active session, redirecting them to the login page.

  Detailed Process

  a. Checks that the session exists
     - Deletes the session if it exists
     - Exits the method otherwise

  b. User is redirected to the login page

- ***delete_player*()** - Deletes the player’s account and all database entries where their ID is referenced. Also deletes the users session and re-directs them to the login page.

  Detailed Process

  a. Get the player_id of the user who made the request (stored in session)
  b. The player is deleted from the Players table as well as the Bag linked to the player_id
  c. The server then deletes the User’s session
  d. User is redirected to the login page

- ***player_home()*** - Finds the player in the database and displays their information on the player home page.

  Detailed Process

  a. Finds the player in the database and retrieves their information (wins, losses, email)
  b. The Player’s home page is rendered and their information is displayed

GameController.rb

In all game controller methods, whenever a change is made to an instance variable within one of the modelled application objects, this change should be saved within the database should it be an instance variable that has a field within the database. This is done to ensure proper game progression (phase progressing as well as gameboard updates).

Pre-Conditions
All functions have the precondition of running current_player() from the application controller. This is what allows for player database lookup to be possible (using the player_id).

Methods

- **find_match() -** This function is used to let players find a match to play a game with. Note that the player should be subscribed to the action cable GameUpdates channel for the game that they are joining.

  Detailed Process

  a. The second a player enters matchmaking, it should search the database for the first available game in the database that does not have two players linked to it and link that player who is searching.
     - If a game is found that only has 1 player the game controller will use action cable to send a message to both players via the GameUpdates channel and redirect them both to /newGame.
       - This includes the initial player waiting on the matchmaking page.
  b. If all games in the Games database table are already linked to two players, a new Game will be created and inserted into the Games table.
     - This new game will be linked to the player who is matchmaking via the game_id.
     - Render a static HTML page that says “Matchmaking...” (see web pages for design).
         - This page should only be displayed if the player does not immediately get placed in a game and redirected as described under step ‘a’ above.
         - Player should be subscribed to the GameUpdates action cable message channel

    - Once another player has entered matchmaking and has been assigned to the same game, the game controller will use action cable to send a message to both players via the GameUpdates channel and redirect them to /newGame.

- ***place_piece*****(x:int, y:int)** - When a player places a piece, this function will be called and access the pieces in the player's bag as well as the gameboard. The objects will then be created based on the database entries and the backend of the system (application functions) will be used to carry out the placement of a piece on a provided intersection(x,y coordinate).

  Detailed Process

  a. Create game, player, and referee objects, as well as all the objects that compose these things from the database tables (joins will be necessary)
  b. Convert the x and y values in the request into integers ranging from 0-6 (see application class details for more information)
  c. The referee will determine if the placement location is valid
  d. After validation the player whose turn it is will place the desired piece on the gameboard
  e. Check to see if the placement resulted in a mill
     - If the move did result in a mill return a message to the player asking them to capture a piece and rending the capture interface of the Game Page
     - If no mill was created have the game proceed to the next turn
  f. Check if phase has changed from place to move
     - Send a message via action game to the players detailing the move, and render the game page (either placement or movement version based on the response from game.getPhase?())

- ***move_piece*****(from_x:int, from_y:int, to_x:int, to_y:int)** - When a player makes a move this function will be called and access the pieces and gameboard through the rails database. The objects will then be created based on the database entries and the backend of the system (application functions) will be used to carry out the movement of a piece.

  Detailed Process

  a. Create game, player, and referee objects, as well as all the objects that compose these things from the database tables (joins will be necessary)
  b. Convert the x and y values in the request into integers ranging from 0-6 (see application class details for more information)
  c. The referee will determine the type of move being made
  d. Based on move type, the referee will validate the move

    - If the move is invalid then send a message back to the player asking them to make a valid move

e. After move validation the player whose turn it is will move the desired piece
f. Check to see if the move resulted in a mill
    - If the move did result in a mill return a message to the player asking them to capture a piece and rending the capture interface of the Game Page
    - If no mill was created have the game proceed to the next turn
        - Send message to player detailing the move, and render the game page with the changes

Pseudocode (One Possible Implementation)

```
// create game from database
// create player from database
// create referee from database

// convert the character x values and integer y values sent in the request to be
integers ranging from 0-6

 move_type = referee.determineMovementType(x, y, piece_x, piece_y)

 If (move_type == move) {
        if(referee.validateMove(x, y, old_x, old_y) == false){
                // invalid move
                // do not progress to next turn
                // Send a message via actioncable the the player that the moves
                was invalid and they should make another move
                // return here
        }
}elsif (move_type == fly) {
        if(referee.validateFly(x, y, old_x, old_y) == false){
                // invalid move
                // do not progress to next turn
                // Send a message via actioncable the the player that the moves
                was invalid and they should make another move
                // return here
        }
}

// if validation was passed, than the player is okay to make their move

player.movePiece() // move the piece on the gameboard
```

```
// once the piece has been moved, we should update the piece in the table to be
linked to the appropriate intersection
// find the piece currently linked to the intersection that was moved from and
update it in the table to be on the appropriate

// check and see if a mill was formed
If ( referee.validateMill() ){
        // a mill was created, send a message telling the player they can capture
        a piece (this will refresh the players UI)
}else{
        // no mill was created, proceed to next player's turn
        game.nextTurn() // updates turn_colour and turn phase

        // if the phase has been updated game should be saved to the game table

        // send a message to the players via action cable, updating their UI and
        presenting the movement phase interface if the phase has changed
}
```

- ***capture_piece(* x:int, y:int *)*** - When a player moves a piece that results in a mill, this function will be called to capture the piece from the board.

    Detailed Process

    a. Create player objects, game object, and referee object from database
    b. Have the referee validate that the coordinates of the capture are valid and the piece to capture is not in a mill when other pieces are available out of a mill
    c. If valid
        - Have the Player capture the piece
        - After the piece is captured the referee checks if the game has been won
            - If the game has been won, have the game object end the game and set the winner
            - Else, have the game proceed to the next turn
                - Send a message detailing the turn to each player to update their game page's
    d. If not valid
        - Message is sent to player who made the capture request saying to choose a valid piece to capture

- ***forfeit()*** - This function will be called to intentionally end the game early, changing game_status to complete and rewarding the win to the opponent.

Detailed Process

a. Create the necessary object instances from the database to construct the game object (game and player and the objects they are composed of)
b. The game must determine which player is the winner (the player whose turn it is not)
c. The player whose turn it is will receive a loss, and the winner will receive a win, once this is done, send a message to the clients via action cable and render the game page (Game Finished version)
    - This should be added to their database fields ‘wins’ and ‘losses’

- ***propose_draw()*** **-** When a player presses the “Propose Draw” button, this function will be called to initiate the start of a draw sequence where a prompt will be sent to the other user to either accept or decline the draw. On draw, change the game_status to complete.

Detailed Process

a. The Game sets the draw flag in itself (proposeDraw()), this will be cleared if the other player declines in the respond_draw request
    - This game update should be saved to the database
b. An alert is sent to the opponent player asking them to accept or decline the draw

- ***respond_draw(String: response)*** **-** This function will be called when a request is made to respond to a draw. It will check and see if the draw flag is set (done by Game), and then check and see if the request was made to accept or decline the draw (also done by Game).

Detailed Process

a. Create an instance of a Game and send the client's response to the Game to process: respondDraw(response)
b. The changes made to Game should then be read by the controller (the winner variable) and saved to the database
c. The controller will check the winner of the game, if ‘none’ the draw was accepted, it will save this to the Game in the database
d. Send a message via action cable to the clients to let them know the game is over and render the game page (Game Finished version)

- ***start_game()*** **-** When a player presses the start game button, the system will start searching for an opponent who is also searching for a game. After matching with another player, the game will assign both players a colour and enter the piece placing phase (the game will start).

Detailed Process

a. Objects for bags, gameboard, players, and game should be created.
    - Bags should be assigned to players

- Players will be used to construct the game
- Pieces should be placed in the bag (however, this is not necessary in this specific implementation as they will just be garbage collected anyways)
    - Instead, simply create pieces in the database Pieces table and link them to their respective Bag via foreign key
- The game object will assign player colours, these colours will be used to determine which colour of pieces should be stored in each players bag

b. The player is redirected to the Game Page and the game has begun

Pseudocode

```
// Create the player objects based on the players in the database (randomly assign each player a colour, normally this would be done in game, but we are doing it in the controller for simplicity)


// create a  new gameboard in the database
// create an array of default empty intersections in the database and link these to the gameboard
// create a new referee
// access currently existing players, link them to the new game in the database via foreign key

// create a gameboard object based on the database entry we just made that will be given to each player (they both play on the same board)
// create new empty bag objects for each player

bag_1 = Bag.new()
bag_2 = Bag.new()
player_1 = Player.new(name_1, bag_1, board)
player_2 = Player.new(name_2, bag_2, board)

//Create a new game object, and give it the new players and referee
game = Game.new(player_1, player_2)
game.setup()

// once setup is complete, the Player objects that we still have access to on top-level can be saved to the database, along with their respective bags
// link the database bags to their respective players

player_1.save
player_2.save
bag_1.save
```

```
bag_2.save

if (player_1.getColour? == white)
    // add 9 white pieces to the database
    // link the database pieces to the players bag
    // Note that the pieces already exist in the bag object, but this will be trash
    collected anyways
Else
    // add 9 black pieces to the database
    // link the database pieces to the players bag
    // Note that the pieces already exist in the bag object, but this will be trash
    collected anyways

// THE SAME IF STATEMENT ABOVE SHOULD BE REPEATED FOR
PLAYER_2

// now that all pieces, bags, players, and the gameboard are ready to go in the
database, we can begin the game

// send a message to the players and let the player who got chosen as white to
make the first move
// Note that we do not want to call nextTurn here as none of the functionality in
that method is desirable (we do not need to update the turn_colour)
```

- ***play_again()*** - Clears the game board and re-routes to /findmatch.

  Detailed Process
  a. The GameBoard is cleared
  b. The client should be re-routed to the matchmaking queue to search for another match

Rails - URL Calls

The formatting of URL calls is as follows:

/url (*url parameters if they are needed*) to: *controller#method*

1. **GET**
    a. /login **to: sessions#check_session (This is the root address)**
    b. /home **to: player#player_home**
    c. /logout **to: player#logout_player**
2. **POST**
    a. /findmatch **to:game#find_match**
    b. /newgame **to: game#start_game**
    c. /playagain **to: game#play_again**
    d. /signup **to: player#create_account**
    e. /loginplayer **to: sessions#create_session**
3. **PUT / PATCH**
    a. /move (from_x:int, from_y:int, to_x:int, to_y:int) **to: player#move_piece**
    b. /capture (x:int, y:int) **to: player#capture_piece**
    c. /place (x:int, y:int) **to: player#place_piece**
    d. /forfeit **to: game#forfeit**
    e. /draw **to: game#propose_draw**
    f. /drawrespond(response: String) **to: game#respond_draw**
4. **DELETE**
    a. /delete **to: player#delete_player**

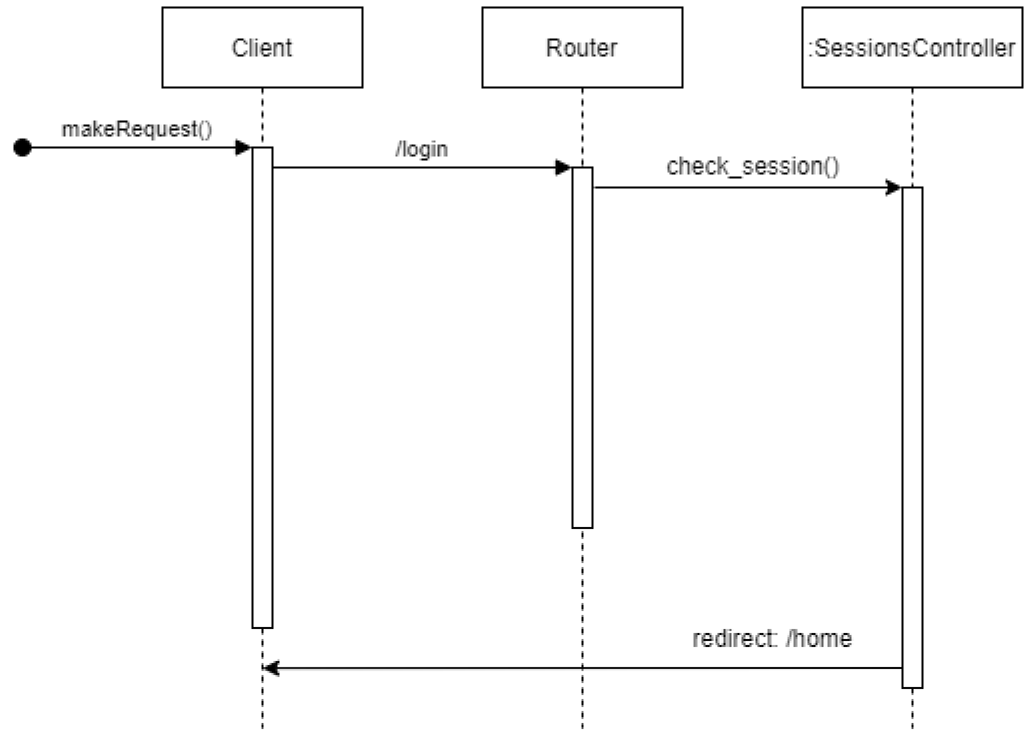
Rails - Client Server Sequence Diagrams

In these sequence diagrams, the client starts by making a request. We chose to present the diagrams this way rather than having the client be completely external to provide clarification on where the URL call is coming from. This also simplifies the understanding of how the controller is reached from the client-side of the server, without mentioning any web page/browser specifics. **MakeRequest()** is simply a representation of the client making a request to the server.

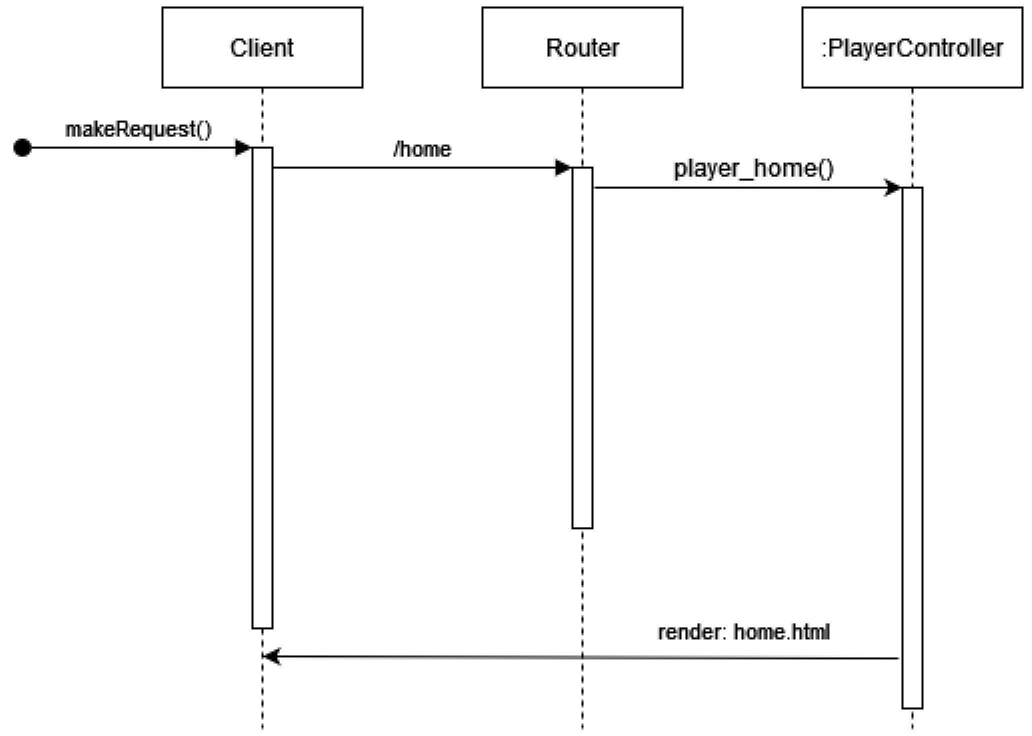
Action Cable is the architecture used during game-based functions to update the client's game information. This can be seen as an arrow connecting back to the client in the diagrams where it is used.

Note that some of these diagrams are quite small in scale and may be hard to view. Full-sized versions can be provided upon request.

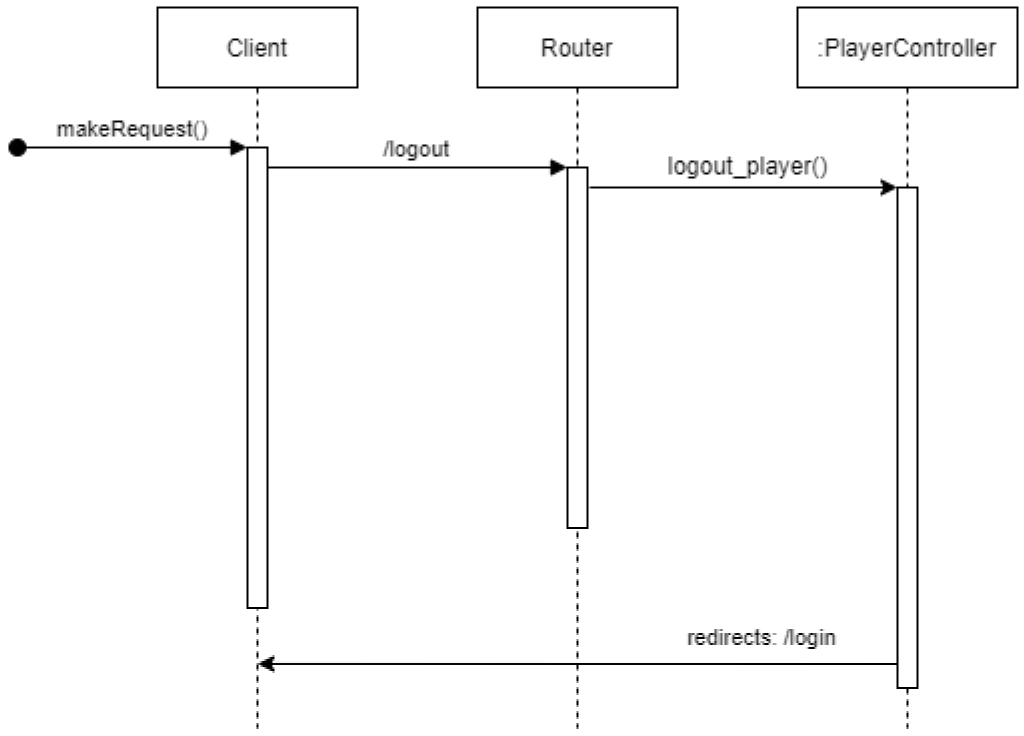
Login (navigate to homepage) - A session exists in this case



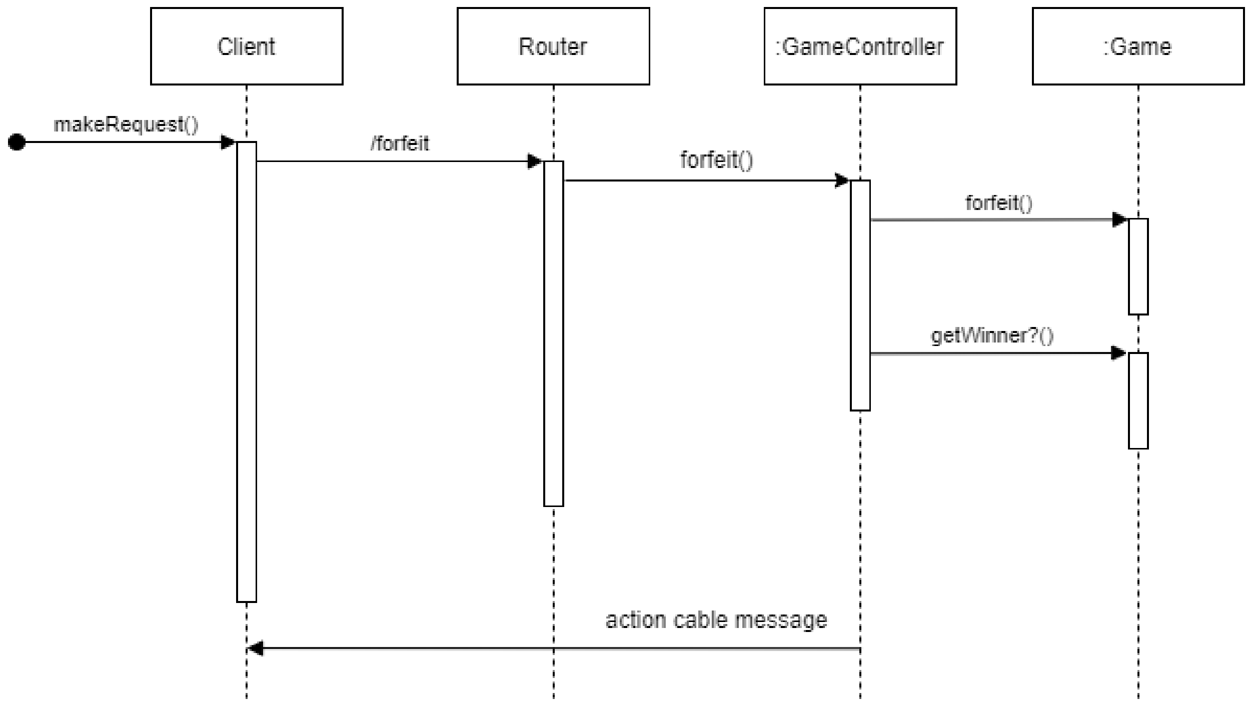
Load the Player Home Page



Logout of Account

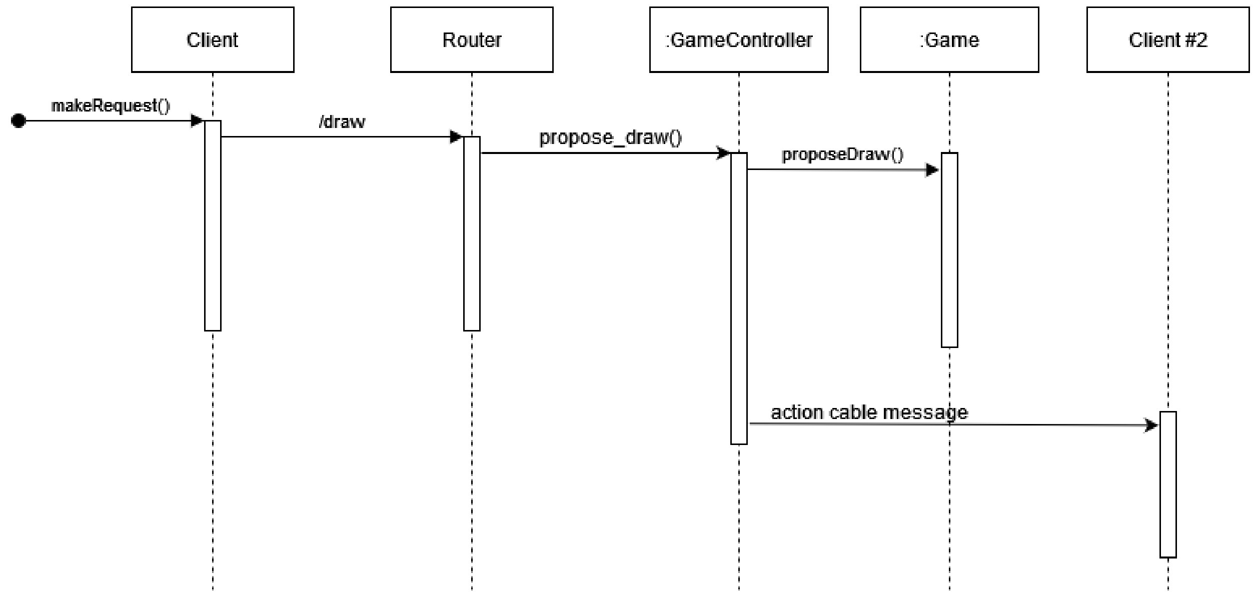


Forfeit the Game



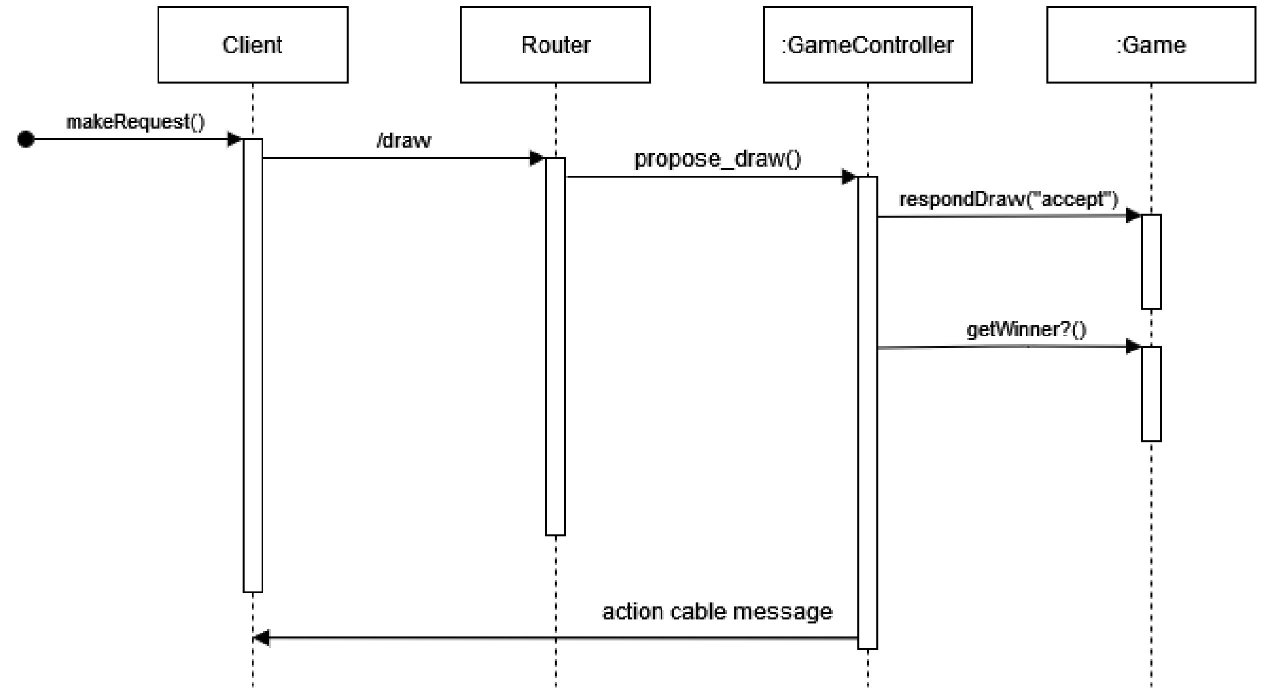
Sends a message via action cable letting the clients know that the game is complete.

Propose a Draw



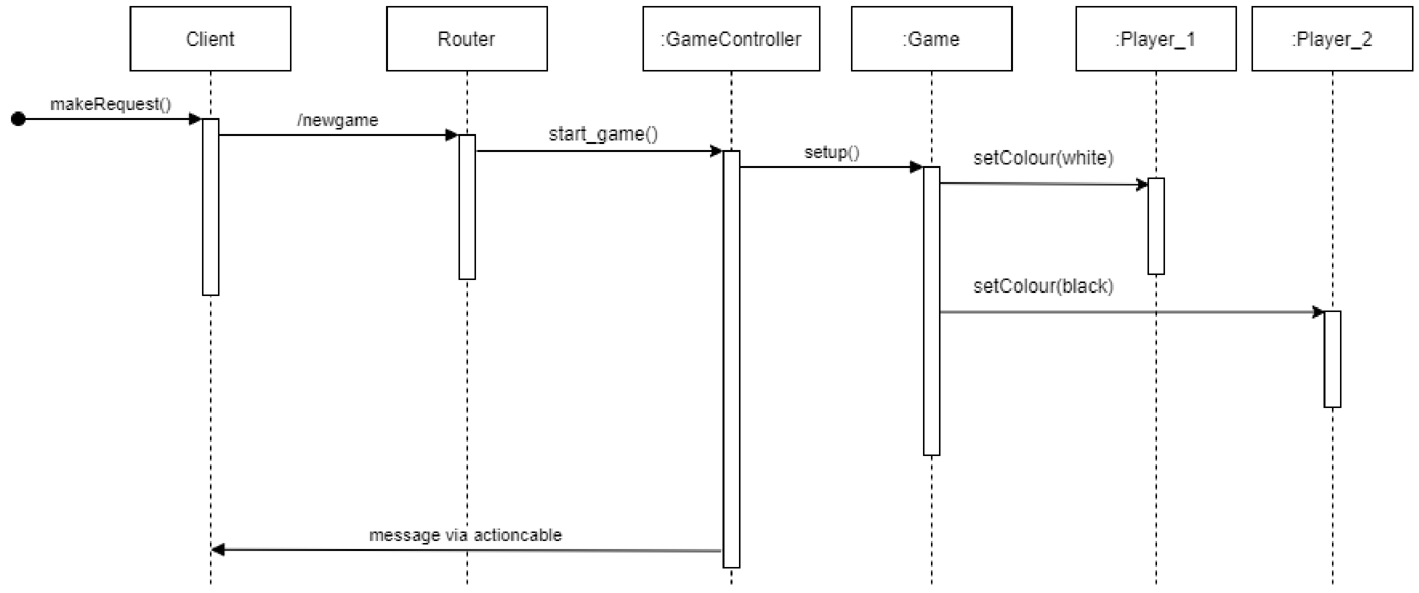
Sends a message via action cable asking the opposing client (player) if they'd like to accept the draw

Respond to a Draw



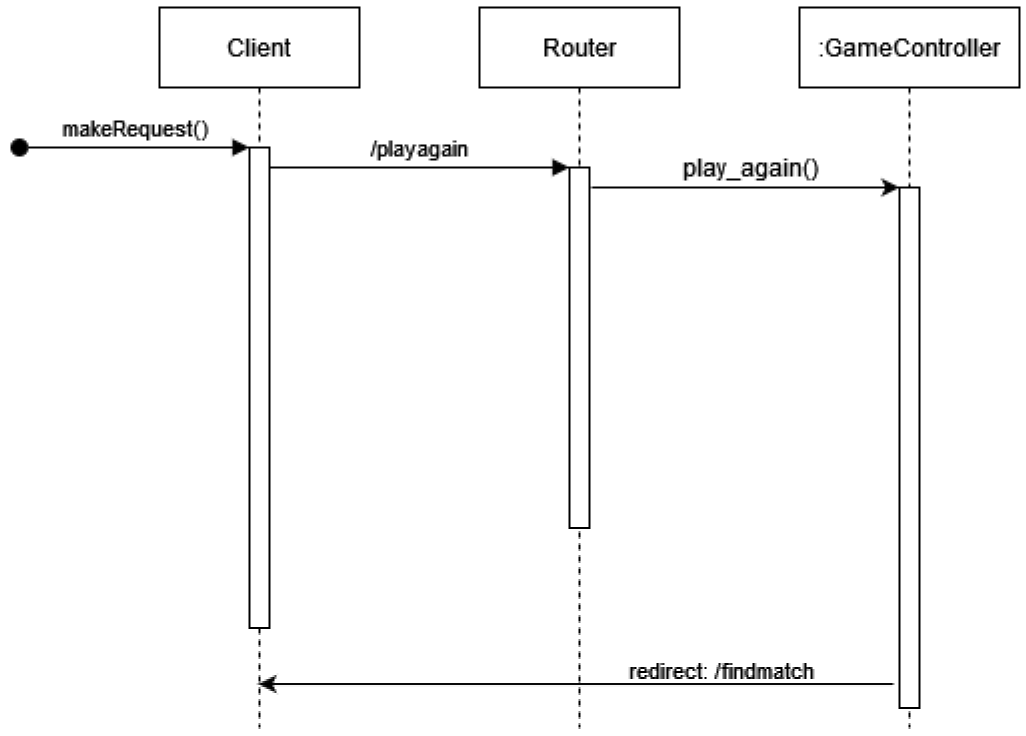
Sends a message via action cable letting the clients know that the game is complete.

Start a New Game

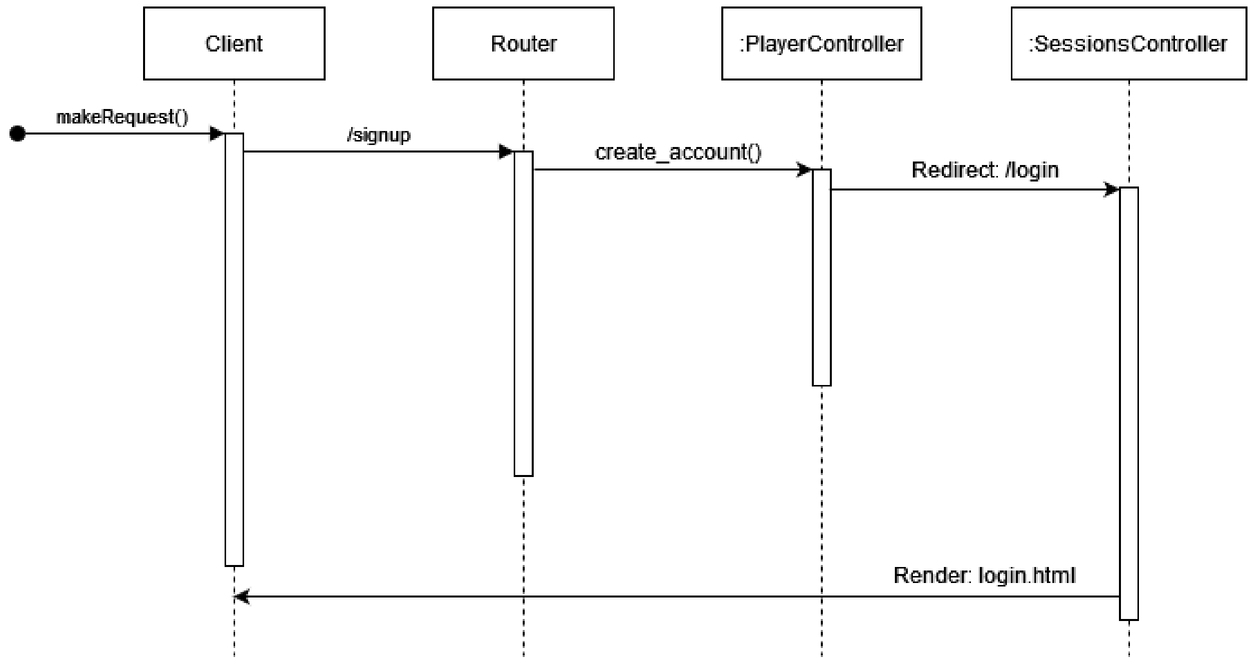


Action cable will send message to client of move result as return value.

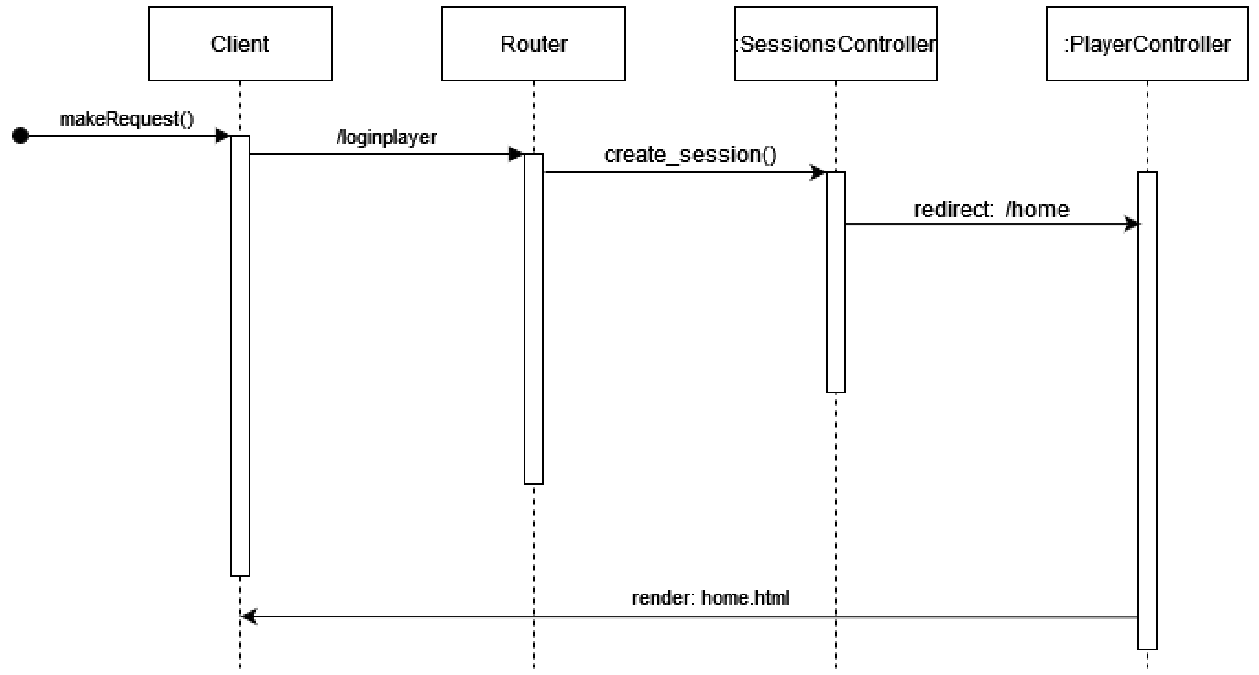
Play Again



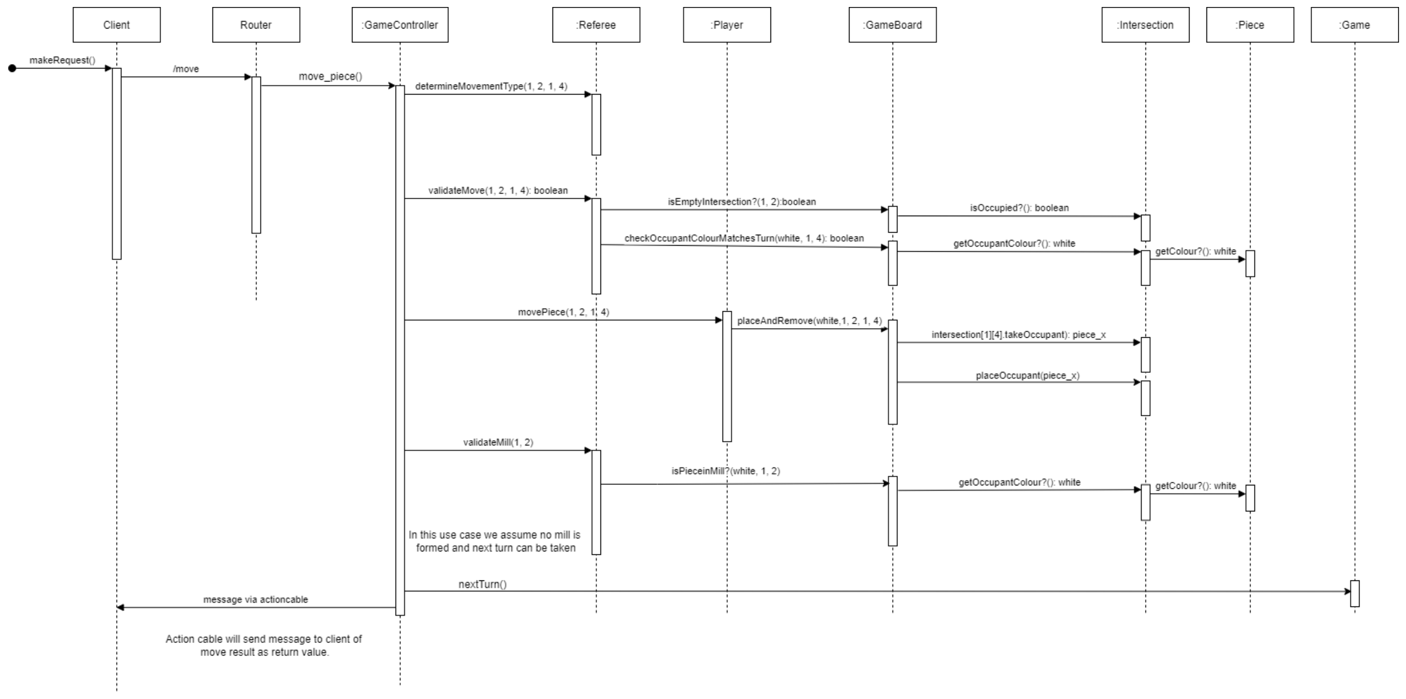
Signup - Create an account



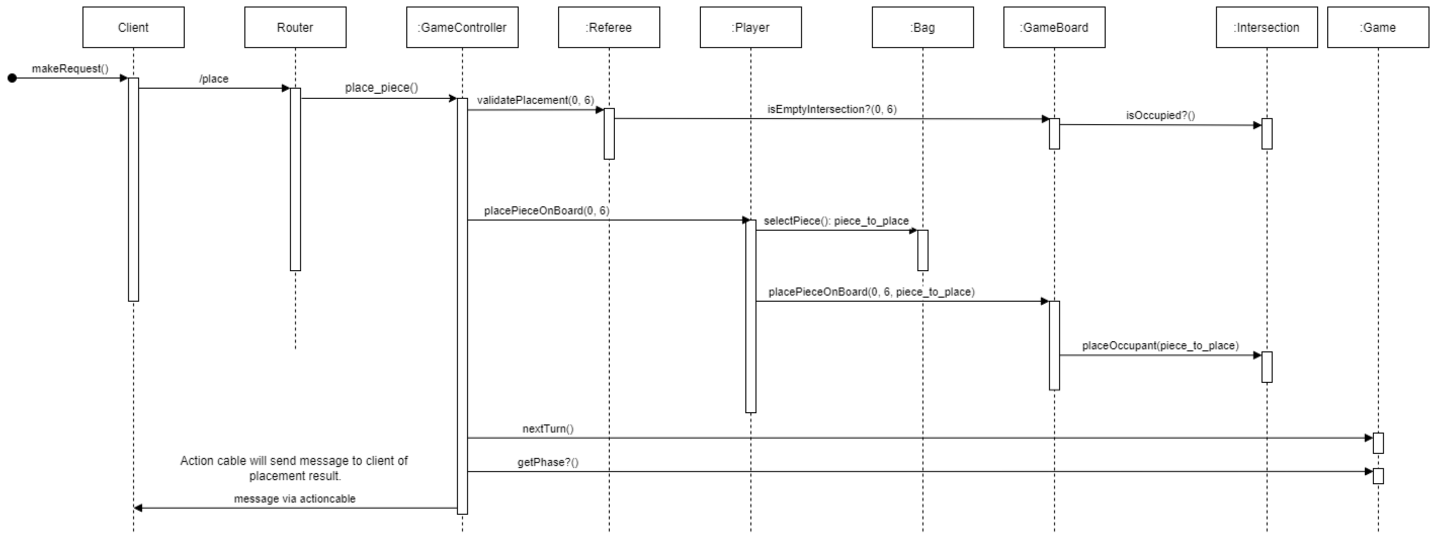
Login Player (happens when login button pressed)



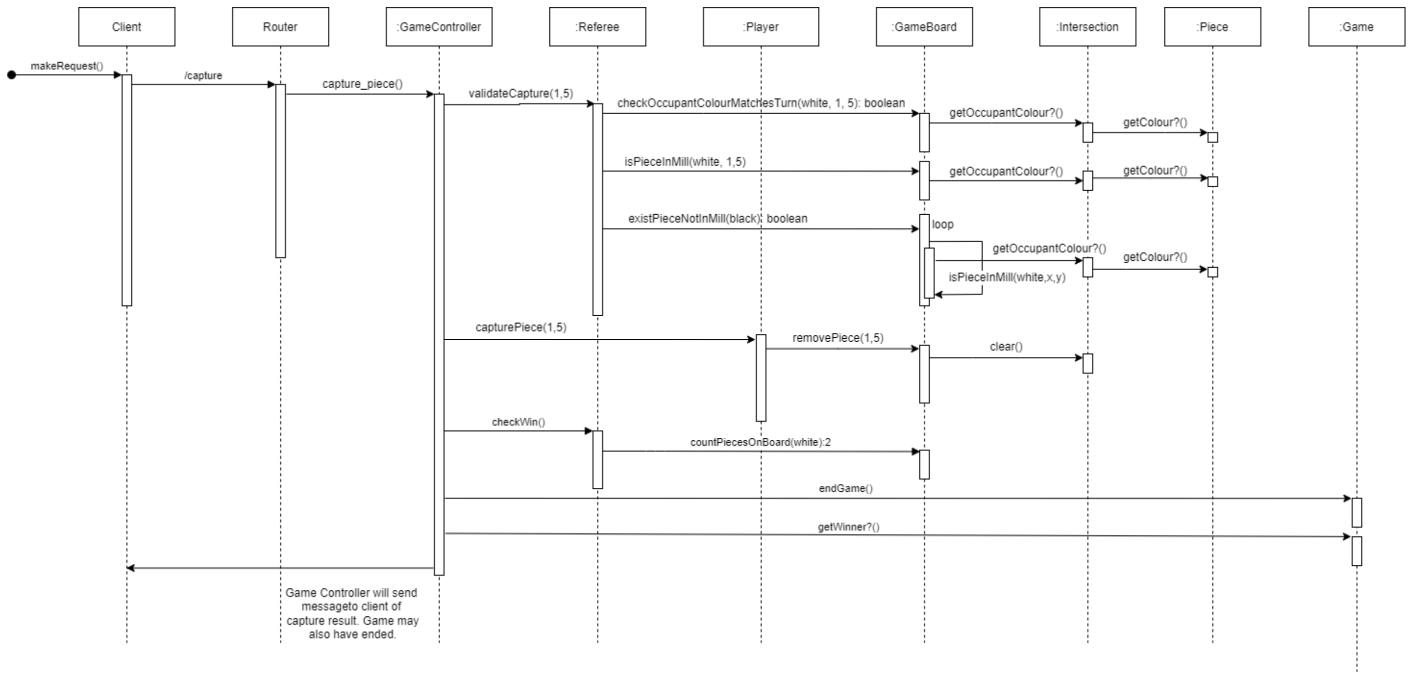
Move a Piece


Client
Router
:GameController
:Referee
:Player
:GameBoard
:Intersection
:Piece
:Game
makeRequest()
/move
move_piece()
determineMovementType(1, 2, 1, 4)
validateMove(1, 2, 1, 4): boolean
isEmptyIntersection?(1, 2):boolean
isOccupied?(): boolean
checkOccupantColourMatchesTurn(white, 1, 4): boolean
getOccupantColour?(): white
getColour?(): white
movePiece(1, 2, 1, 4)
placeAndRemove(white,1, 2, 1, 4)
intersection[1][4].takeOccupant): piece_x
placeOccupant(piece_x)
validateMill(1, 2)
isPieceinMill?(white, 1, 2)
getOccupantColour?(): white
getColour?(): white
In this use case we assume no mill is formed and next turn can be taken
nextTurn()
message via actioncable
Action cable will send message to client of move result as return value.

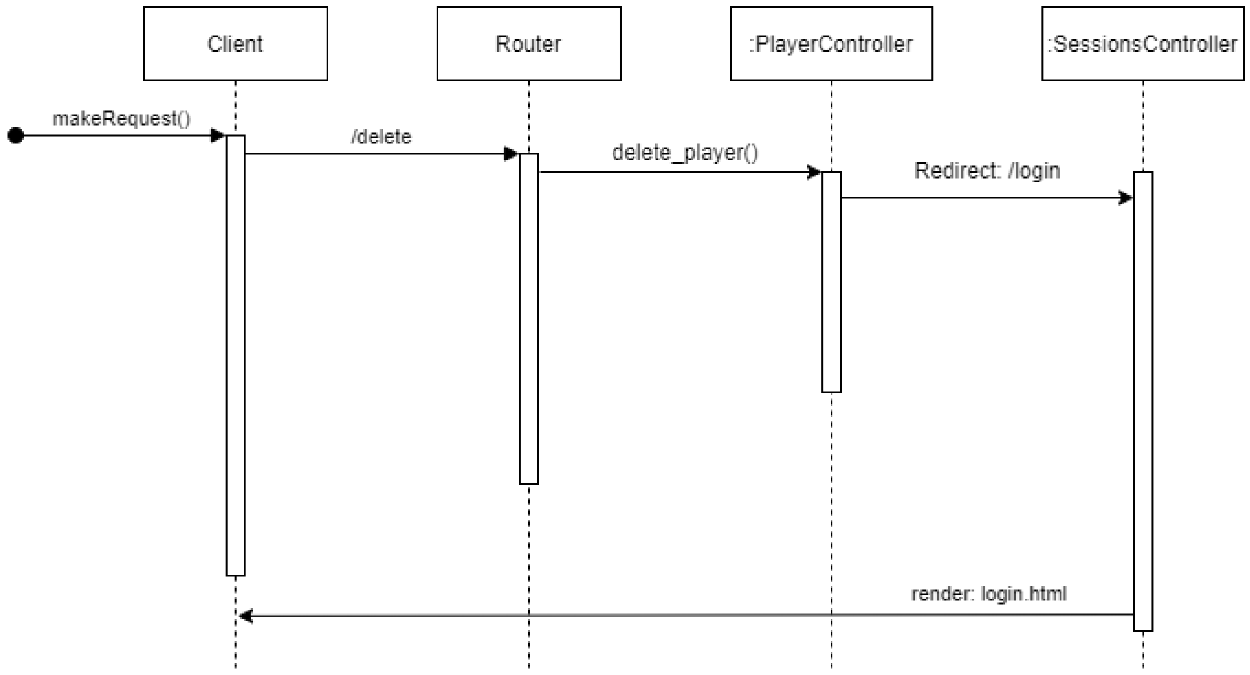
Place a Piece



Capture a Piece

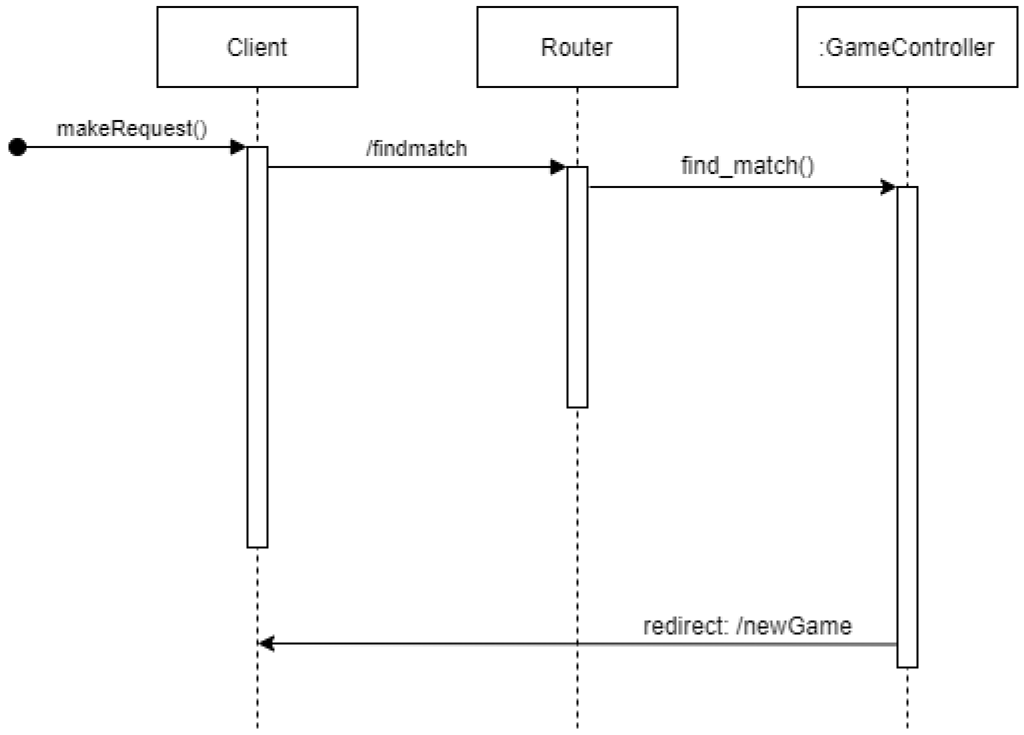


Delete User Account



Matchmaking

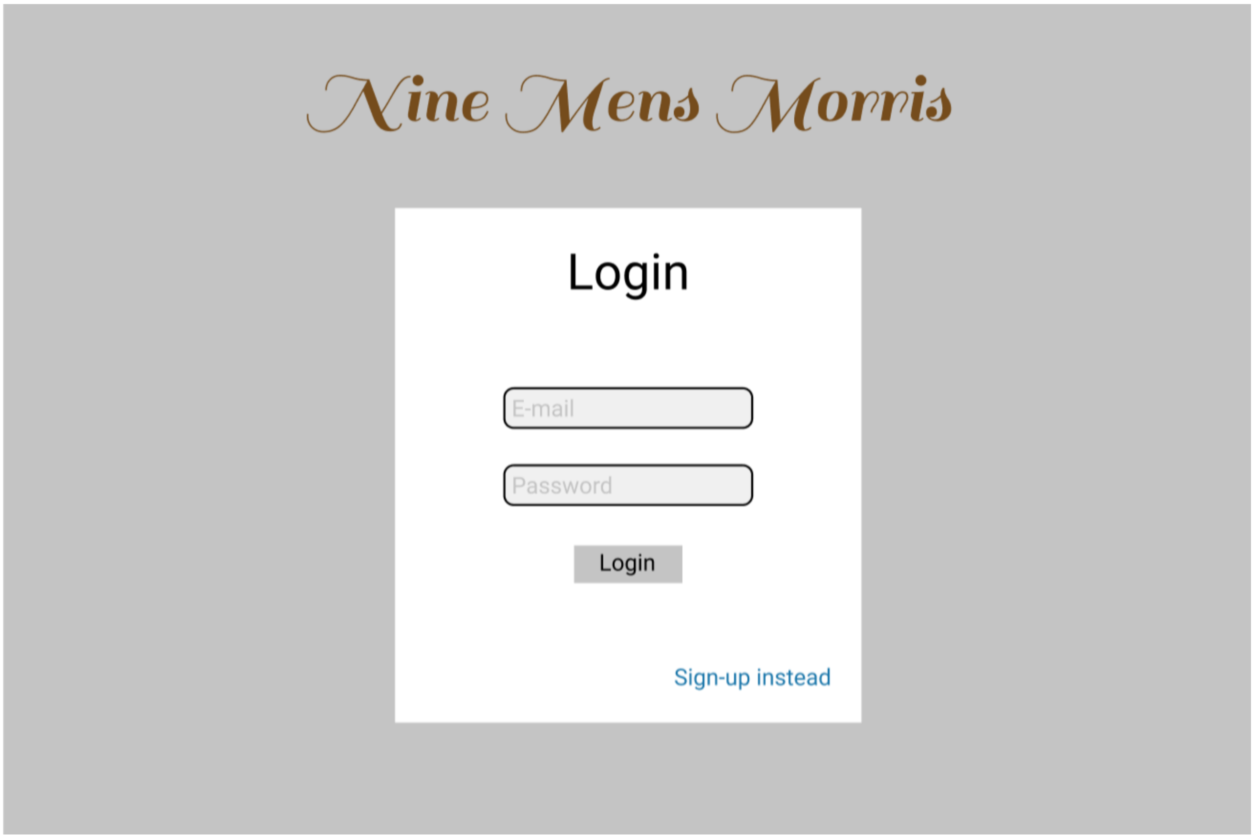
Note: this diagram displays the flow if a game is available to join. In this scenario both clients will be redirected to /newGame and the game will begin. In an alternate scenario where the client is the first one to start matchmaking, the matchmaking.html page would be rendered.



Rails - Web Pages

Login Page (login.html)

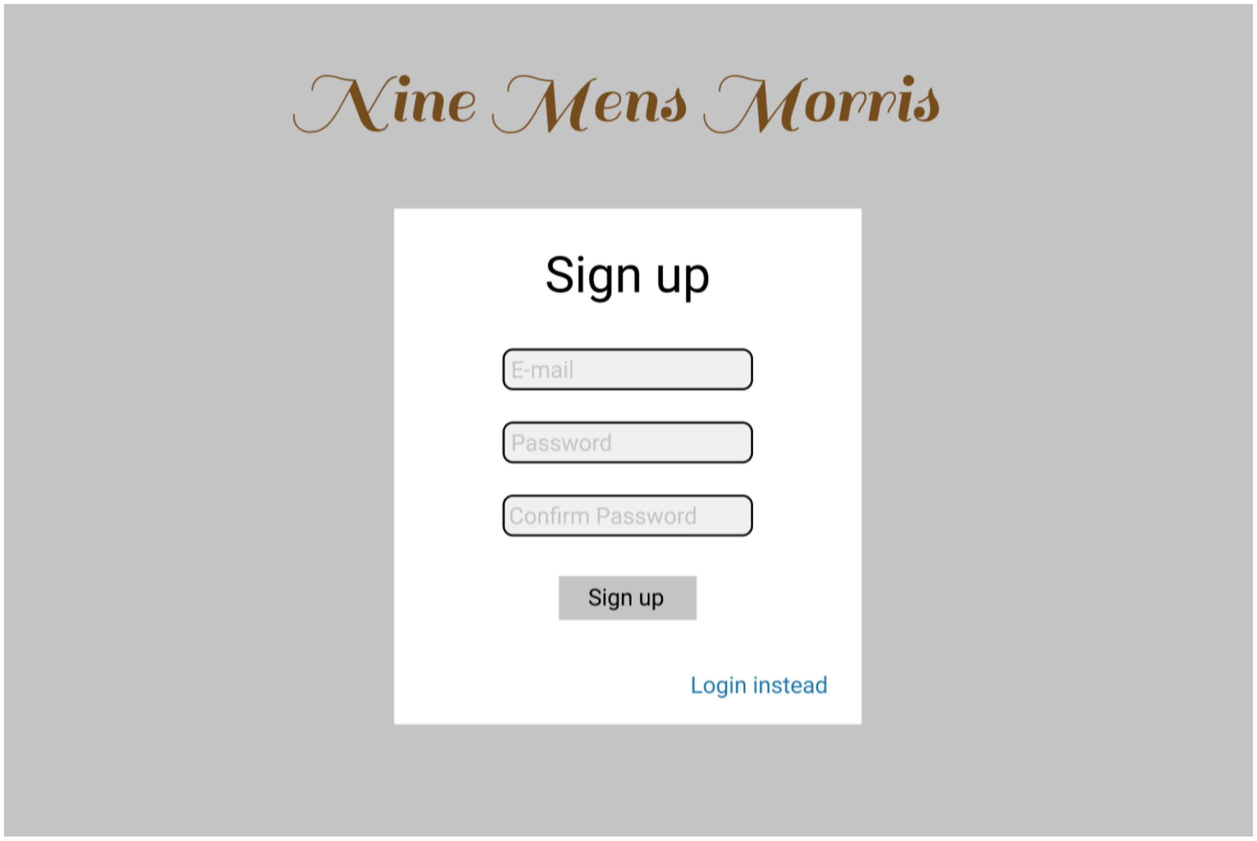
The login page to the website. Lets the user enter an email and password to log in.



- Buttons
    - Login: /loginplayer
- Hyperlink
    - Sign-up instead: /signup

Sign up Page (signup.html)

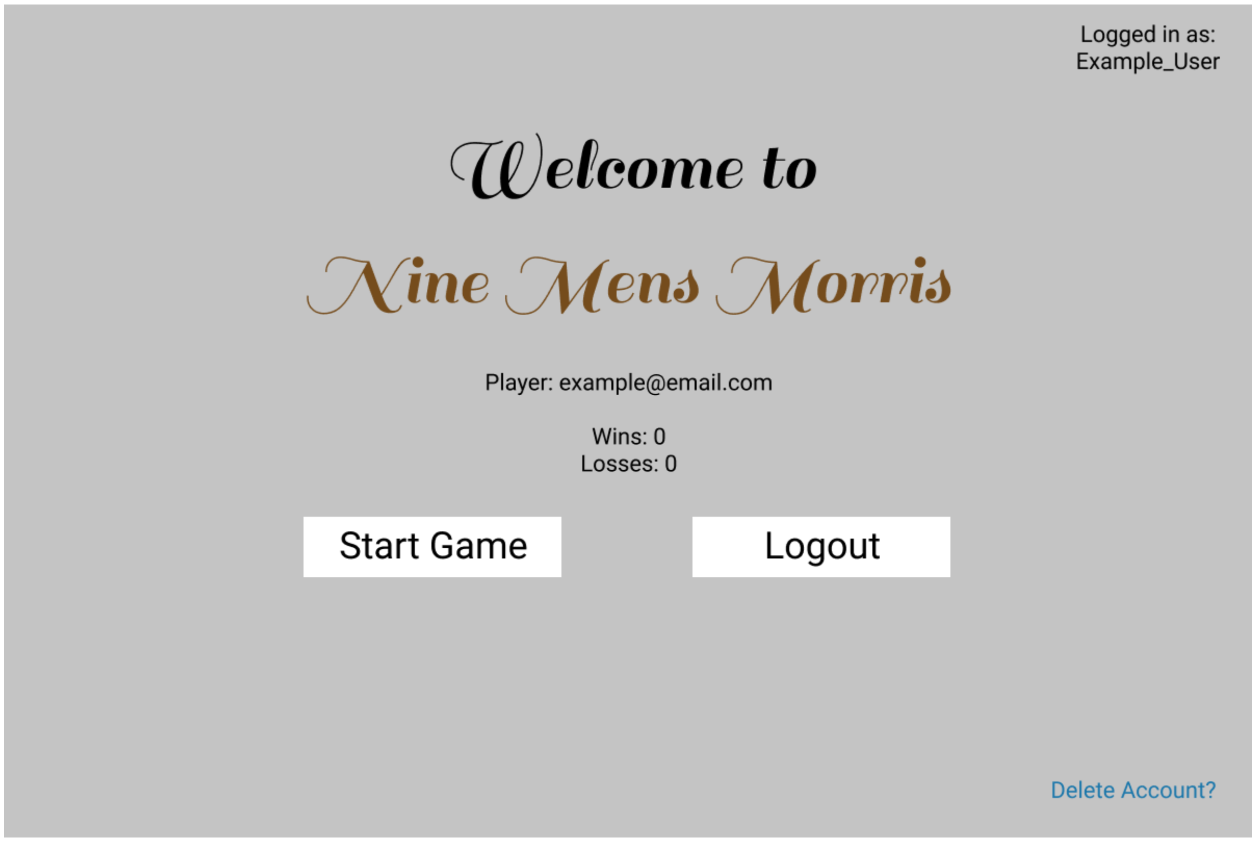
Lets a user sign up for an account on the website.



- Buttons
  - Sign up: /signup
- Hyperlink
  - Login instead: /login
- Form
  - The form fields will be used to create a user account when the sign up button is pressed. This should use the built-in rails form builder.

Player Home Page (home.html)

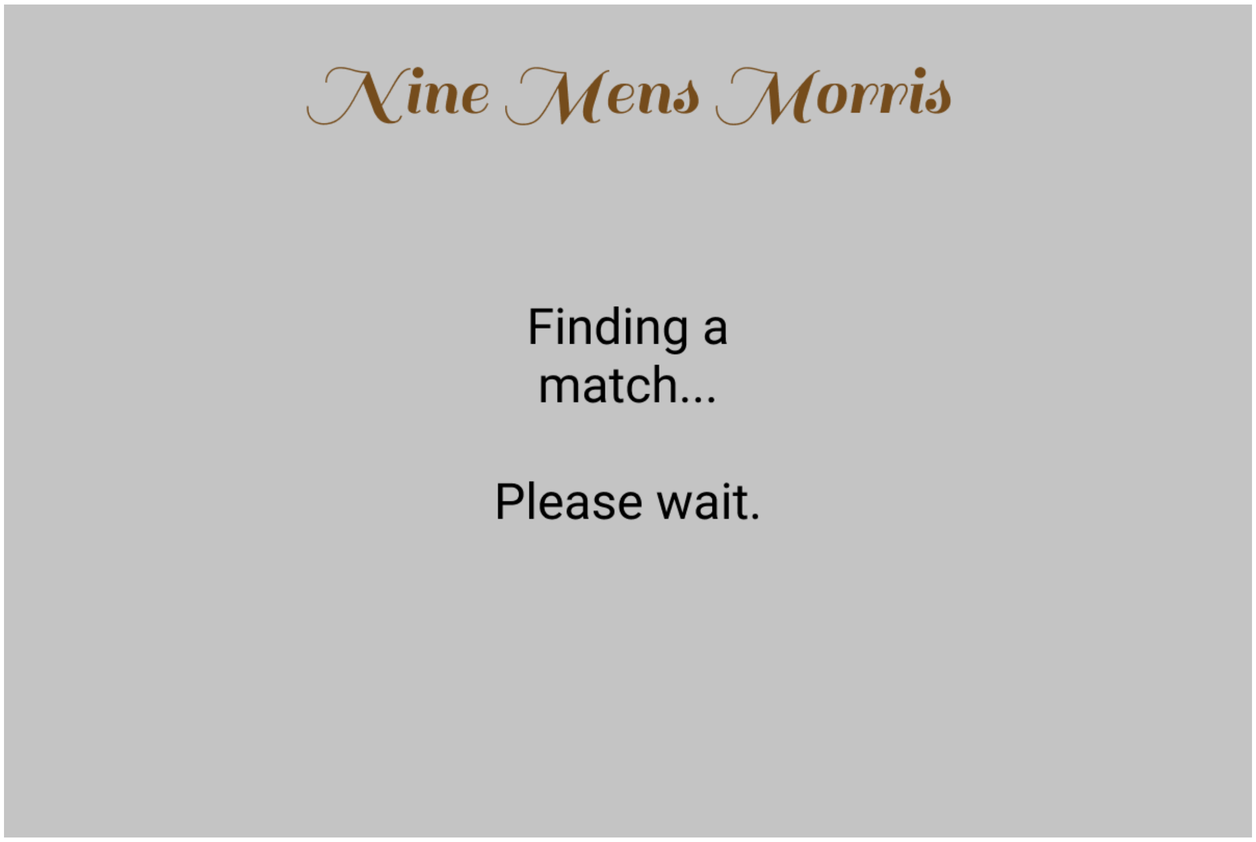
The home page of the website displays a user’s win/loss scores and lets them start a new game or logout.



- Buttons
  - Start Game: /findmatch
  - Logout: /logout
- Hyperlink
  - Delete Account: /delete

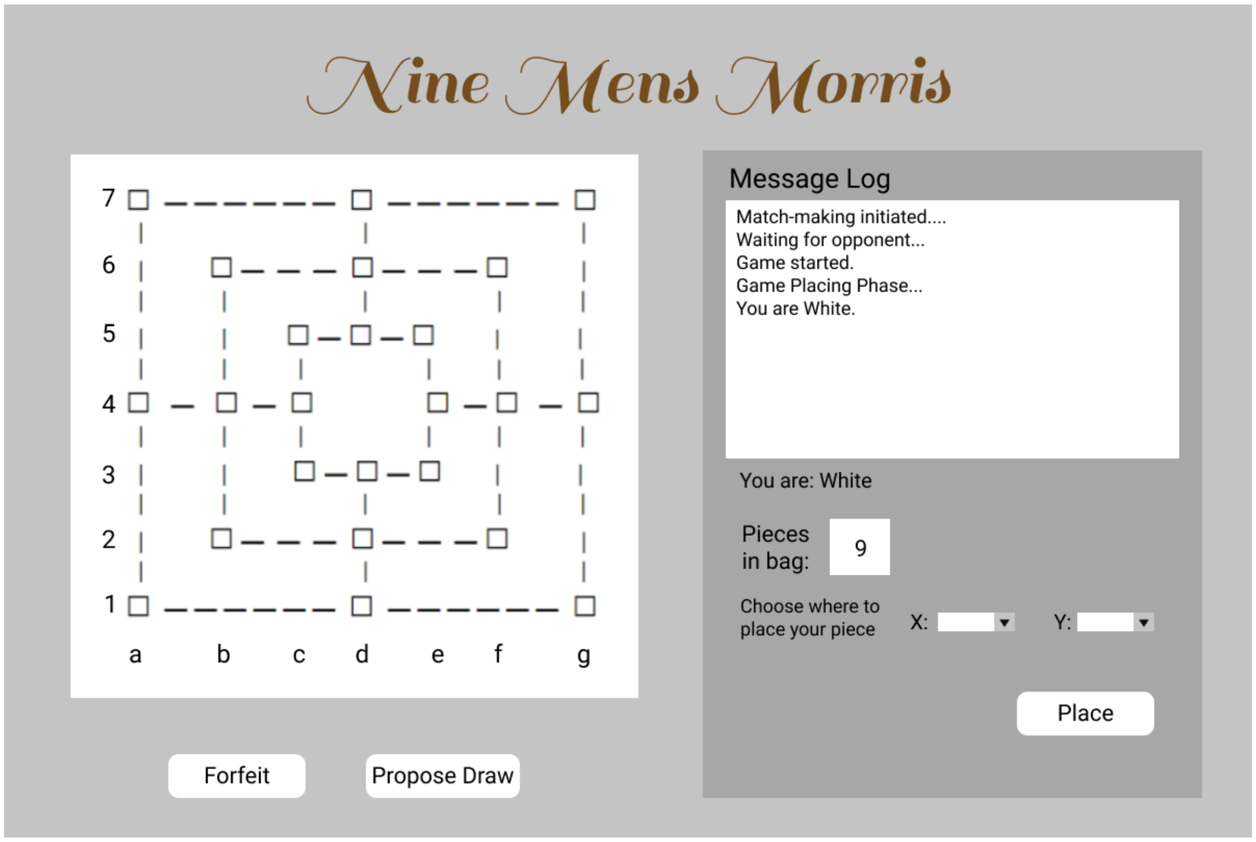
Matchmaking Page (matchmaking.html)

The player will see this page after clicking Start Game. They will stay on this page until another player is found. Once an opponent is found they will navigate to /newGame.



Game Page (game.html) (Placing Phase - Where the game starts)

The game page during the placing phase of the game. Shows the game board, the colour they are playing as, a message log, the number of pieces left a player has to place on the board, a dropdown menu containing the X and Y coordinates of the places on the board they may place their piece, and a button to place down their piece. The player can also forfeit the game or propose a draw using the respective buttons. These two buttons will appear on all Game Page screens except for when the game is finished.



Match-making initiated....
Waiting for opponent...
Game started.
Game Placing Phase...
You are White.

You are: White

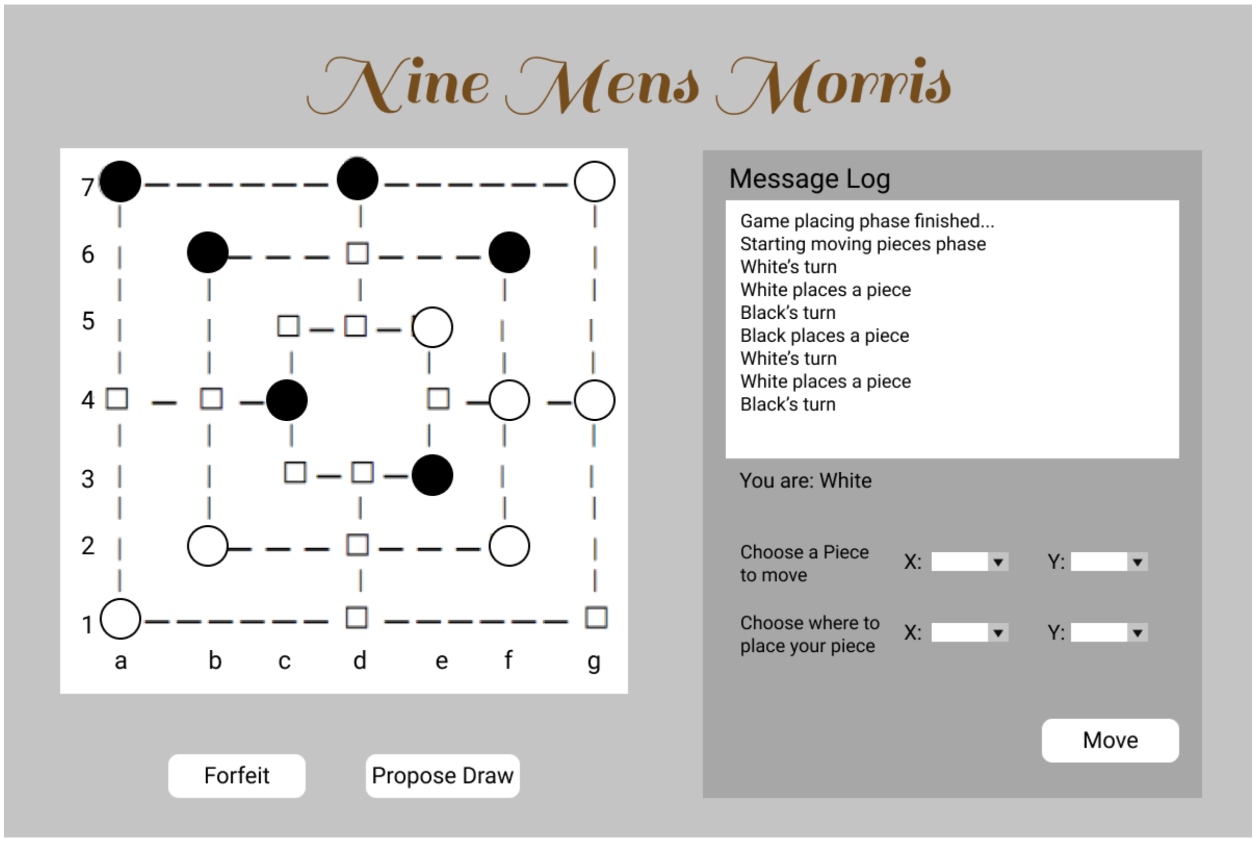
Pieces in bag: 9

Choose where to place your piece

- Buttons
  - Forfeit: /forfeit
  - Propose Draw: /draw
  - Place: /place
- Select Dropdowns
  - Choose a piece to move
    - X: sent as parameter with /place on Place button click
    - Y: sent as parameter with /place on Place button click

Game Page (game.html) (Movement Phase)

Contains similar components as the Placement Phase game page. The board is filled with both the player's and opponent's pieces. The biggest difference now is that the player must first provide the coordinates of a piece they own on the board that they wish to move and the coordinates of the destination in which they wish to move that piece to.

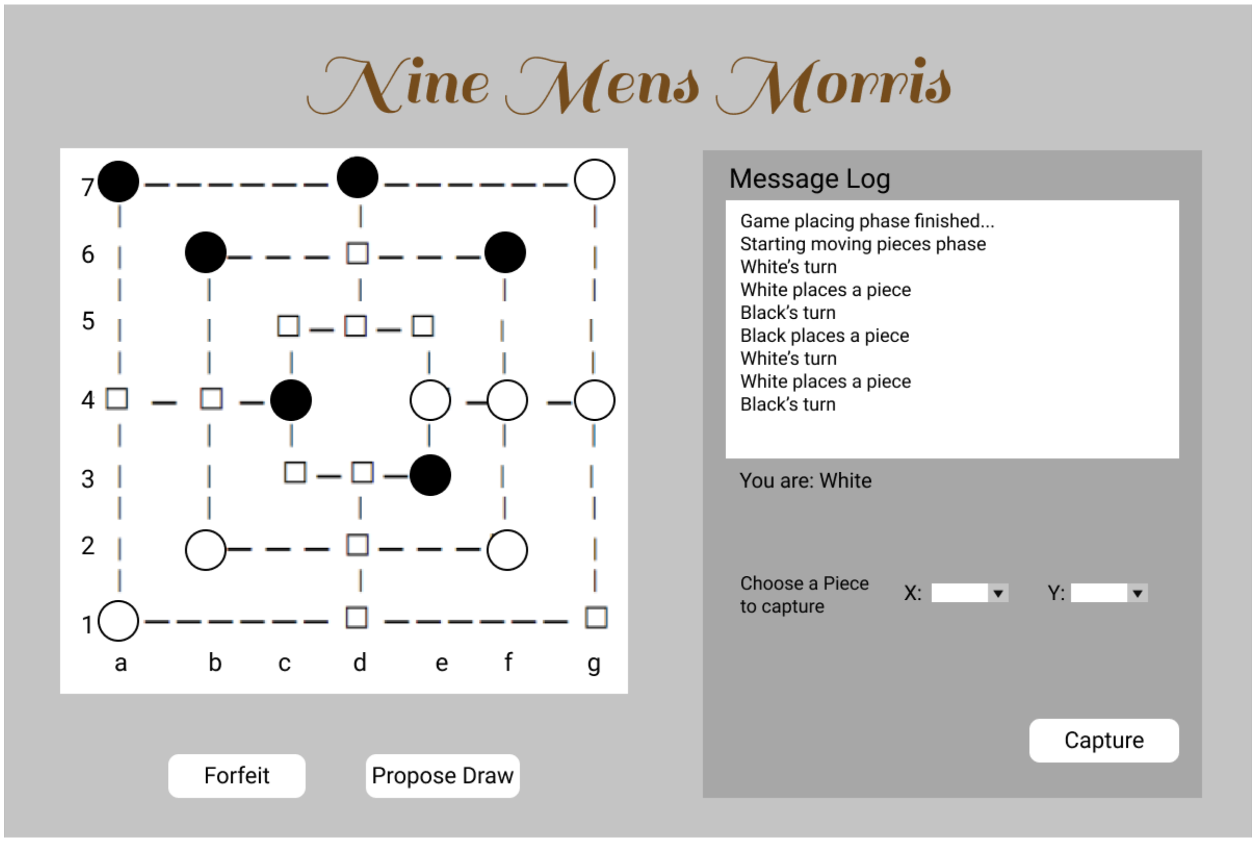


Game placing phase finished...
Starting moving pieces phase
White's turn
White places a piece
Black's turn
Black places a piece
White's turn
White places a piece
Black's turn

- Buttons
  - Forfeit: /forfeit
  - Propose Draw: /draw
  - Move: /move
- Select Dropdowns
  - Choose a piece to move
    - X: sent as parameter with /move on move button click
    - Y: sent as parameter with /move on move button click
  - Choose where to place your piece
    - X: sent as parameter with /move on move button click
    - Y: sent as parameter with /move on move button click

Game Page (game.html) (After forming mill, page will display capture interface)

The game page once a mill is formed. When a mill is formed, the user is presented with this page, allowing them to choose the coordinates of an opponent's piece that they wish to capture.

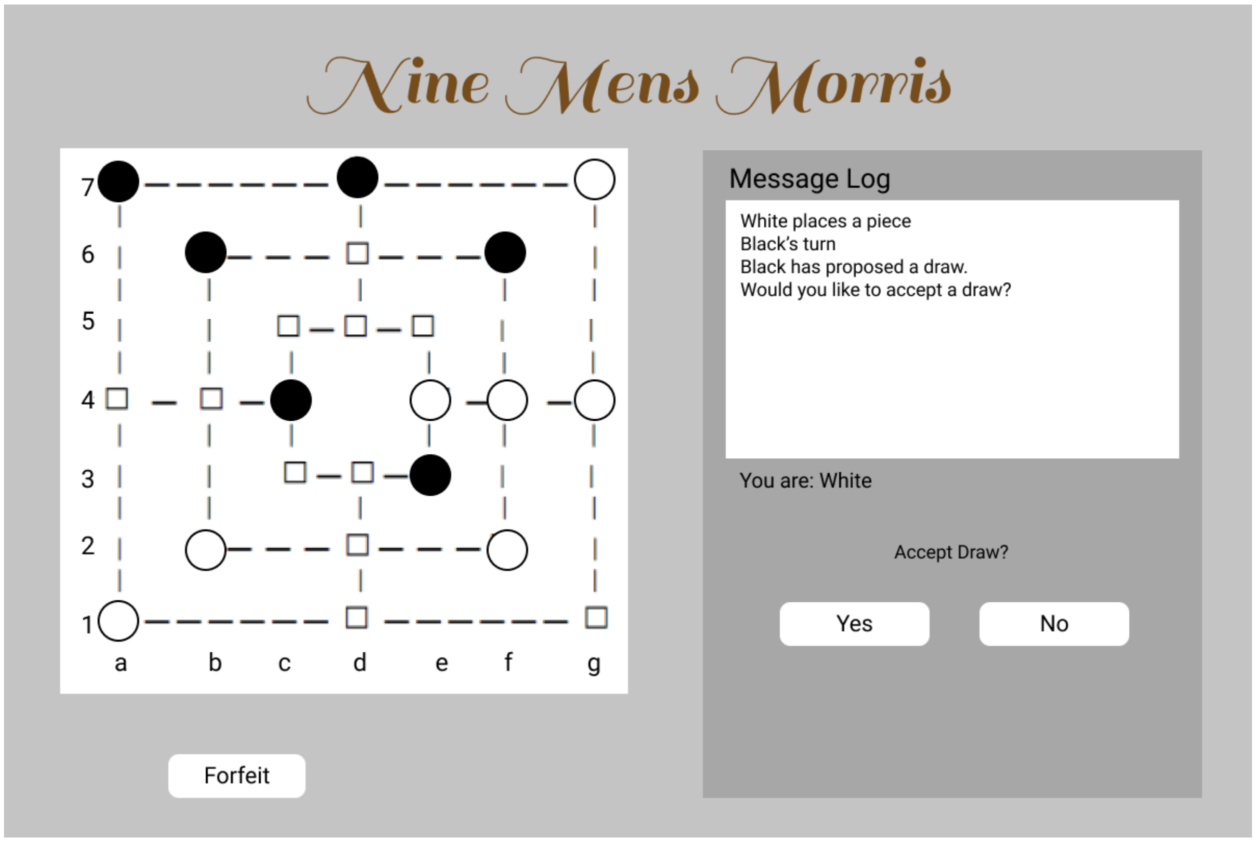


Game placing phase finished...
Starting moving pieces phase
White's turn
White places a piece
Black's turn
Black places a piece
White's turn
White places a piece
Black's turn

- Buttons
    - Forfeit: /forfeit
    - Propose Draw: /draw
    - Capture: /capture
- Select Dropdowns
    - Choose a piece to capture
        - X: sent as parameter with /capture on Capture button click
        - Y: sent as parameter with /capture on Capture button click

Game Page (game.html) (Propose Draw)

The game page for when the opposing player proposes a draw. The player is given the option to accept or decline the draw.



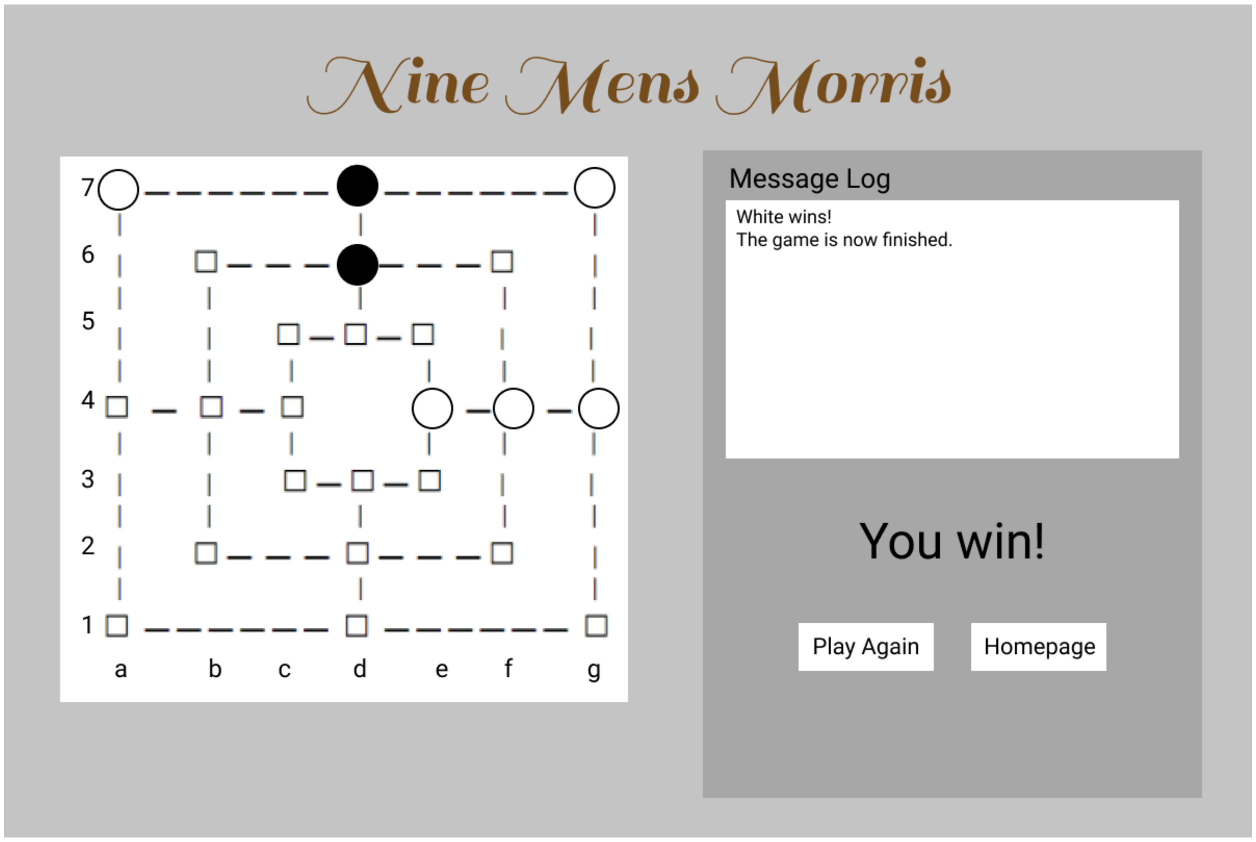
White places a piece
Black's turn
Black has proposed a draw.
Would you like to accept a draw?

You are: White

- Buttons
  - Forfeit: /forfeit
  - Yes: /drawrespond parameter: 'accept'
  - No: /drawrespond parameter: 'decline'

Game Page (game.html) (Game Finished)

The game page for when the game is finished. Shows the player “You Win!” or “You Lose!” depending on the outcome of the game. A player can choose to play another game or head back to the homepage.



White wins!
The game is now finished.

- Buttons
    - Play Again: /playagain
    - Homepage: /home

Note: Under each web page image are the interactable elements on the page as well as which URL Call they should make when used.

List of components for design (please refer to the list below each page to see which URL calls the components on it make):

- Buttons
    - Login
    - Logout
    - Sign up
    - Start Game
    - Forfeit
    - Propose Draw

    - Place
    - Move
    - Capture
    - Play Again
    - Homepage
    - Yes
    - No
- Dropdown Menus
    - Placing pieces phase
        - X
        - Y
    - Choosing piece of move
        - X
        - Y
    - Choosing where to place piece
        - X
        - Y
    - Capture piece
        - X
        - Y
- Hyperlink
    - Sign-up Instead
    - Login Instead
    - Delete Account

Unicode characters used:

⚪ - U+26AA

⚫ - U+2B24

☐ - U+2B24

— - the “em dash”, but Hyphens can be used as well

“Nine men’s Morris” title font: **Sail font type**

Appendix A

Algorithm to Search for a Mill - This is baseline code, but can be used as an example of how to check different sections of the board for a mill. It is not guaranteed to be free of errors, and can be coded using a more efficient algorithm if desired. If you have any questions please contact the design team.

Note that GameBoard 'intersections' is an object array sized [7][7]
The goal is to check if piece at x,y is in a mill, either horizontally or vertically.

```
isPieceInMill(colour, x, y)
// note that an edge case is required for column D and row 4 where it is possible for 6 pieces to exist in
each instead of 3

piece_in _vertical_mill = true
piece_in _horizontal_mill = true

// if in the middle column check the edge case in where column = D (3 in the intersections array)
if  (x == 3){
        // we need to check the 2 sides of the centre separately as two 3 piece mills can be formed in this
        column

        // if the y value is less than 3 we only need to check the lower half of the board
        if(y < 3){
                // this checks for a vertical mill on the bottom half of the board
                for (int i = 0; i < 3; i++){
                        // we want to skip all spaces in the array where there is no object present, this
                        happens due to the odd position of intersections on the board
                        if( self.intersection[x][i] != null){
                                if( intersection[x][i].getOccupantColour? != colour ){
                                        // if we find an occupant that doesn't match the player's colour
                                        than it isn't possible for the piece to be in a vertical mill

                                        piece_in _vertical_mill = false
                                }
                        }
                }

        }else{          // this checks for a vertical mill on the upper half of the board
                for (int i = 4; i < 7 ; i++){
                        if( self.intersection[x][i] != null){
                                if( intersection[x][i].getOccupantColour? != colour ){
                                        piece_in _vertical_mill = false
                                }
                        }
                }
        }
        // check the horizontal mill for the given piece, which half of the board it is on does not
        matter
        for (int i = 0; i < 7; i++){
                // checks all possible horizontal intersections and ignores null ones
                if( self.intersection[i][y] != null){
                        if( intersection[i][y].getOccupantColour? != colour ){
                                piece_in _horizontal_mill = false
                        }
                }
        }

// if in the middle row check the edge case in where row = 4 (3 in the intersections array)
}else if (y == 3){
        // this checks for a horizontal mill on the left half of the board
        if(x < 3){
                for (int i = 0; i < 3; i++){
                        if( self.intersection[i][y] != null){
                                if( intersection[i][y].getOccupantColour? != colour ){
                                        piece_in _horizontal_mill = false
                                }
                        }
                }
        }else{          // this checks for a horizontal mill on the right half of the board
                for (int i = 4; i <= 6; i++){
                        if( self.intersection[i][y] != null){
                                if( intersection[i][y].getOccupantColour? != colour ){
                                        piece_in _horizontal_mill = false
                                }
                        }
                }
        }

        // check the vertical mill for the given piece, which half of the board it is on does not matter
        for (int i = 0; i < 7; i++){
                // checks all possible horizontal intersections and ignores null ones
                if( self.intersection[x][i] != null){
                        if( intersection[x][i].getOccupantColour? != colour ){
                                piece_in _vertical_mill = false
                        }
                }
        }

// if it is in neither of these edge cases a normal search of adjacent spaces can be done
}else{

        // check for any possible vertical mill
        for (int i = 0; i < 7; i++){
                // checks all possible horizontal intersections and ignores null ones
                if( self.intersection[x][i] != null){
                        if( intersection[x][i].getOccupantColour? != colour ){
                                piece_in _vertical_mill = false
                        }
                }
        }

        // check for any possible horizontal mill
        for (int i = 0; i < 7; i++){
                // checks all possible horizontal intersections and ignores null ones
                if( self.intersection[i][y] != null){
                        if( intersection[i][y].getOccupantColour? != colour ){
                                piece_in _horizontal_mill = false
                        }
                }
        }

}
```

// now that we have checked for all mills we must check both boolean variables to see if either is still true. If one of them is true then the piece is confirmed to be in either a horizontal or vertical mill and we should return true

```
If (piece_in _horizontal_mill == true || piece_in_vertical_mill == true){
        Return true
} else{
        Return false
}
```