Designer’s Information:

Dixant Patel
Ryan Paul
Nareshri Babu
Jarryl Subang
Siri Chandana Kathroju

Name of Program:

Nine Men’s Morris Web Application

Table of Contents

Executive Summary

Game Description

Nine Men's Morris is a ***zero sum strategy game*** that dates back to as early as 1400 BC. This is a two player game and is played on a board that has 24 intersections. The objective of this game is the win by eliminating opponent pieces off the board until there are only 2 opponent pieces remaining.

High Level Design Overview

Our design for **Nine Men's Morris** uses an object oriented approach to represent the various components of the game. A typical game of Nine Men's Morris has a board, 2 players, each player has 9 men and a container to hold their pieces. In our design, we represented these objects as various classes that interact together to allow the users to play the game. Each of these classes contain methods which perform a single responsibility related to the class object. Through the use of instance variables and class methods, the classes interact with other classes (illustrated through the use of arrows in our class diagram). Unlike an actual in person game of nine men's morris, our version of the game contains a database system which holds key information about the game play which gets updated as needed. Finally, the design document for the game Nine Men's Morris contains a user interface of what the game should look like including components such as buttons, forms/textboxes, etc.

Limitations

Our design does not take into account the possibility of a draw happening. Although a draw is possible through perfect play, it is very unlikely for two players to end the game with a draw.

Things to Look Out for in the Design

For our design we envisioned having a separate Referee class which handles any rule violations that might occur. Although this logic could have been included in the Board class or the Game class, having a separate class gives a cleaner logic. (Following the logic of the Class Diagram will highlight the benefits of the Referee class)

Furthermore most of our complicated logic lies in the Board class. Having a good understanding of the methods in this class will allow you to implement the logic easily. (Reading the method description will help you get a better idea for the algorithms to implement)

Game Rules

Game Description

The game consists of ***2 Players***, ***18 Men*** (pieces) and a board with ***24 Intersections****.* The goal of this game is to win by eliminating the opponents pieces by creating a **Mill** to the point where they only have **2 Men** remaining on the board (see figure 2.0) or by blocking them so they have no valid moves remaining (see figure 2.1). Each player is given 9 pieces of a different colour(i.e. white or black). The player who plays white is given the advantage of the first move. A player can create a **Mill** by placing 3 pieces of their own in a row or column *(i.e see figure 2.2)*. When a player creates a Mill, they must remove an opponent's men off the board. The players are **NOT** allowed to remove a man that is in a mill unless only 3 men remain (i.e see figure 2.3). A player is only able to move to an empty adjacent intersection on the board. However, the player is able to ***fly*** when they only have 3 pieces remaining. A fly is when a player can move its men to any location on the board that is not occupied.

For a more detailed explanation on the rules check out this site:
https://www.mastersofgames.com/rules/morris-rules.html

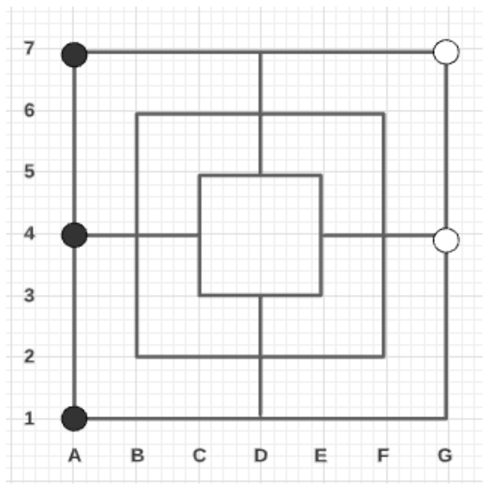


Figure 2.0: *2 pieces remain [White]*

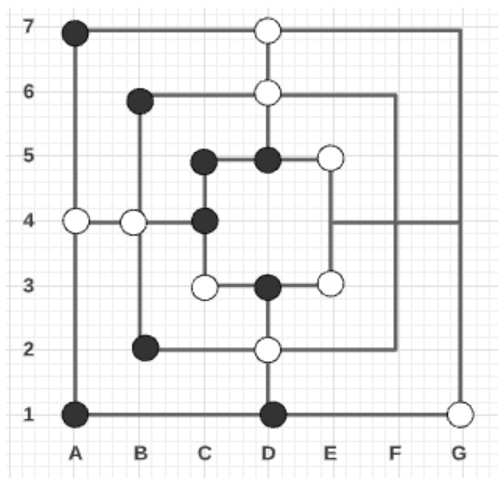


Figure.2.1: *White Blocking Black*

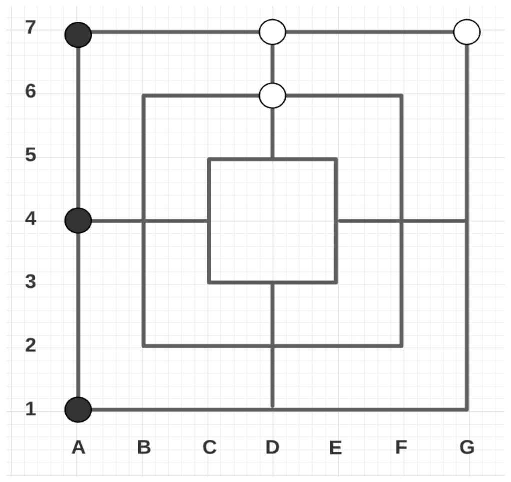


Figure 2.2: *A **mill** created by black*

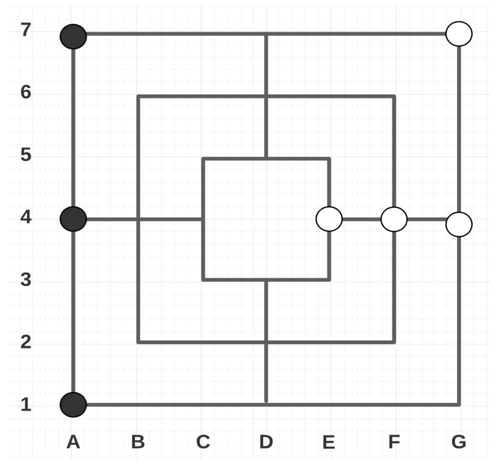


Figure 2.3: *White is allowed to remove from black's mill*

Use Cases

Use Case 1: Set up

Use Case Name: Set up the board
Primary Actor: Player
Goal - description of use case purpose: To initiate the game of Nine Men's Morris
Stakeholders List: Player 1 and Player 2
Initiating Event: Two players agree to play Nine Men's Morris
Main Success Scenario:

1. Player 1 requests Player 2 to a game of Nine Men's Morris
2. Player requests Game to select order of Players

3. Game receives request and selects order of Player
4. Player requests Bag of Men from Game and Game receives request
5. Game generates Bag and sends it to Player and Game begins

Post Conditions: The game has begun
Alternate flows or exceptions:
N/A
Use cases Utilized:
N/A
Scenario Note:
- **Player 1**: White
- **Player 2**: Black
- **Mill**: Three of one player's pieces (men) lined up horizontally/vertically
- "Fly" Rule: When a player is down to 3 pieces, that player may move to any empty space
- A player must move a piece on every turn
- The player wins if their opponent has no valid moves OR has only 2 pieces left

Use Case 2: Remove

Use Case Name: Remove an opponent piece off the board.
Primary Actor: Player
Goal: To remove an opponent piece off the board
Stakeholders List:
Player wants to remove the opponent piece off the board
Initiating Event: Player has two pieces in a row with an available spot that is adjacent to its current position to place a third piece to make a mill.
Main Success Scenario:
1. Player selects a spot to move to that is adjacent to its current position and has two of its own pieces in a horizontal or vertical row
2. Player places its own piece on an adjacent intersection on the board to make a horizontal or vertical "mill" with its own two other pieces.

3. Player selects an opponent piece to be removed
4. Player removes the opponent's piece off the board

The opponent player receives the message and they remove the selected piece as per the message and the game continues
Post Conditions: The opponent piece must be removed from the board
Alternate flows or exceptions:
2a. Player removes opponent piece by creating a mill by flying to an available spot
 2a.1. Player selects for a spot to move that has two of its own pieces in a horizontal or vertical row
 2a.2. Player places its own piece on that intersection to make a horizontal or vertical "mill" with its own two other pieces
 2a.3. Player selects an opponent piece to be removed
 2a.4. Player removes the opponent's piece off the board
7a. Players cannot remove an opponent piece off a mill
 7a.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
 7a.2. Player places its own piece on that intersection to make a horizontal or vertical "mill" with its own two other pieces

7a.3. Player selects an opponent piece to be removed
7a.4. Player attempts to remove the opponent’s piece off the board that is already in a “mill” but can’t
7a.5. Player selects and removes another opponent’s piece that is not in a “mill” and game continues

7b. Players can remove a piece from a mill if there are no other pieces available.
7b.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
7b.2. Player places its own piece on that intersection to make a horizontal or vertical “mill” with its own two other pieces
7b.3. Player selects an opponent piece to be removed
7b.4. Player attempts to remove the opponent’s piece off the board that is already in a “mill” since no other pieces are available to be removed
7b.5. Player removes the opponent’s piece and the game continues

10a. Player removes one of three opponent pieces off the board and game ends
10a.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
10a.2. Player places its own piece on that intersection to make a horizontal or vertical “mill” with its own two other pieces
10a.3. Player selects an opponent piece to be removed
10a.4. Player removes one of three opponent’s piece off the board (leaving only two opponent pieces on the board); game ends

Use Case 3: Move

Use Case Name: Player moves their piece
Primary Actor: Player
Goal: Move Piece
Stakeholder List: Player 1 / Player 2
Initiating Event: Players turn is initiated.

Main Success Scenario:
1. Player selects their man to move
2. Board highlights adjacent move locations for selected man
3. Player selects one of the highlighted spaces to move their man
4. Player's man moves to the selected space and game continues

Post Conditions:
- Other player's turn begins

Alternate Flow / Extensions:
2a. Player only has 3 men left on the board
2a.1. Board highlights every empty intersection on the board
2a.2. Player selects one of the highlighted spaces to move their man
2a.3. Player's man moves(flys) to the selected space and game continues

4a. Player's man moves to the selected space and mill is created
4a.1. Player selects one of the opposing player's men to remove from the board
4a.2. The board removes the selected man from the game and game continues

4b. Player's man leaves a mill, destroying it, then moves to the selected space
    4b.1. The board sets the Man to be removable from the board.
    4b.2. The board sets the other former Mill members to be removable from the board.

Use Cases Utilized:
- Use Case 2: Remove a piece

Scenario Note:
- **Player 1:** White.
- **Player 2:** Black
- **Mill:** Three of a players pieces placed adjacent to one another
- **Fly:** Placing a piece on any free spot on the board

Use Case 4: End Game by removing a piece

Use Case Name: End game by removing a piece
Primary Actor: Player
Goal: To win the game
Stakeholders List:
Player 1 and Player 2
Initiating Event: Player moves a piece and creates a Mill
Main Success Scenario:
1. Player selects an opponent piece to be removed
2. Player removes the opponent's piece off the board
3. Game verifies that only 2 opponent pieces remain on the board
4. Game declares the winner and loser

Post Conditions: Game ends and the winning Player is declared.
Alternate flows or exceptions:
N/A
Use Cases Utilized:
- Use Case 2: Remove a piece
- Use Case 3: Create a Mill

Scenario Note:
- **Player 1:** Black
- **Player 2:** White
- **Mills:** Three of one player's pieces (men) lined up horizontally/vertically

Use Case 5: End Game with no valid moves

Use Case Name: End Game with no valid moves
Primary Actor: Player
Goal: To win the game by leaving opponent with no valid moves
Stakeholders List:
Player 1 and Player 2
Initiating Event: Player places a piece on a intersection
Main Success Scenario:
1. Board verifies move

2. Game verifies that Player 2 is blocked with no more valid moves available
3. **3.** Game declares the winner and loser

Post Conditions: Game ends and Player wins the opponent has no valid places to move their pieces to
Alternate flows or exceptions:
1a. Player 1 has made an illegal move
    1a.1. Board verifies move
    1a.2. Board requests Player to do another move
    1a.3. Player selects a valid man to move
2a. Player makes a perfect play
    2a.1. Game ends in draw with perfect play
Use cases Utilized:
- Use Case 5: move piece
- Use Case 3: create a Mill

Scenario Note:
- **Player 1:** Black
- **Player 2:** White
- **Mills:** Three of one player's pieces (men) lined up horizontally/vertically

Remote Use Cases

Use Case 1: Set up

Use Case Name: Set up the board
Primary Actor: Player
Goal - description of use case purpose: To initiate the game of Nine Men's Morris
Stakeholders List: Player 1 and Player 2
Initiating Event: Two players agree to play Nine Men's Morris
Main Success Scenario:
6. Player 1 requests Player 2 to a game of Nine Men's Morris
7. Both Players requests Game for empty board and Game receives request
8. Game generates board and sends it to each Player
9. Player requests Game to select order of Players
10. Game receives request and selects order of Player
11. Player requests Bag of Men from Game and Game receives request
12. Game generates Bag and sends it to Player and Game begins

Post Conditions: The game has begun
Alternate flows or exceptions:
N/A
Use cases Utilized:
N/A
Scenario Note:
- **Player 1**: White
- **Player 2**: Black
- **Mill**: Three of one player's pieces (men) lined up horizontally/vertically
- "Fly" Rule: When a player is down to 3 pieces, that player may move to any empty space

- A player must move a piece on every turn
- The player wins if their opponent has no valid moves OR has only 2 pieces left

Use Case 2: Remove Piece

Use Case Name: Remove an opponent piece off the board.
Primary Actor: Player
Goal: To remove an opponent piece off the board
Stakeholders List:
Player wants to remove the opponent piece off the board
Initiating Event: Player has two pieces in a row with an available spot that is adjacent to its current position to place a third piece to make a mill.
Main Success Scenario:

5. Player selects a spot to move to that is adjacent to its current position and has two of its own pieces in a horizontal or vertical row
6. Player places its own piece on an adjacent intersection on the board to make a horizontal or vertical "mill" with its own two other pieces.
7. Player composes message to the opponent player describing the location of the move
8. Player communicates message to the other player and Board saves state to the database
9. The opponent player receives the message and they update their board as per the message
10. Player selects an opponent piece to be removed
11. Player removes the opponent's piece off the board
12. Player composes message to the opponent player describing the location of the man that is to be removed
13. Player sends a message to the other player via the means of the internet and Board saves the state to the database.
14. The opponent player receives the message and they remove the selected piece as per the message and the game continues

Post Conditions: The opponent piece must be removed from the board
Alternate flows or exceptions:
2a. Player removes opponent piece by creating a mill by flying to an available spot

2a.1. Player selects for a spot to move that has two of its own pieces in a horizontal or vertical row
2a.2. Player places its own piece on that intersection to make a horizontal or vertical "mill" with its own two other pieces
2a.3. Player composes message to the opponent player describing the location of the move
2a.4. Player sends message to the other player and Board saves state to the database
2a.5. The opponent player receives the message and they update their board as per the message
2a.6. Player selects an opponent piece to be removed
2a.7. Player removes the opponent's piece off the board
2a.8. Player composes message to the opponent player describing the location of the man that is to be removed
2a.9. Player sends message to the other player and Board saves state to the database
2a.10. The opponent player receives the message and they remove the selected piece as per the message and the game continues

7a. Players cannot remove an opponent piece off a mill

- 7a.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
- 7a.2. Player places its own piece on that intersection to make a horizontal or vertical “mill” with its own two other pieces
- 7a.3. Player composes message to the opponent player describing the location of the move
- 7a.4. Player sends message to the other player and Board saves state to the database
- 7a.5. The opponent player receives the message and they update their board as per the message
- 7a.6. Player selects an opponent piece to be removed
- 7a.7. Player attempts to remove the opponent’s piece off the board that is already in a “mill” but can’t
- 7a.8. Player selects and removes another opponent’s piece that is not in a “mill”
- 7a.9. Player composes message to the opponent player describing the location of the man that is to be removed
- 7a.10. Player sends message to the other player and Board saves state to the database
- 7a.11. The opponent player receives the message and they remove the selected piece as per the message and the game continues

7b. Players can remove a piece from a mill if there are no other pieces available.

- 7b.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
- 7b.2. Player places its own piece on that intersection to make a horizontal or vertical “mill” with its own two other pieces
- 7b.3. Player composes message to the opponent player describing the location of the move
- 7b.4. Player sends message to the other player and Board saves state to the database
- 7b.5. The opponent player receives the message and they update their board as per the message
- 7b.6. Player selects an opponent piece to be removed
- 7b.7. Player attempts to remove the opponent’s piece off the board that is already in a “mill” since no other pieces are available to be removed
- 7b.8. Player removes the opponent’s piece
- 7b.9. Player composes message to the opponent player describing the location of the man that is to be removed
- 7b.10. Player sends message to the other player and Board saves state to the database
- 7b.11. The opponent player receives the message and they remove the selected piece as per the message and the game continues

10a. Player removes one of three opponent pieces off the board and game ends

- 10a.1. Player selects a spot to move that has two of its own pieces in a horizontal or vertical row
- 10a.2. Player places its own piece on that intersection to make a horizontal or vertical “mill” with its own two other pieces
- 10a.3. Player composes message to the opponent player describing the location of the move
- 10a.4. Player sends message to the other player and Board saves state to the database
- 10a.5. The opponent player receives the message and they update their board as per the message
- 10a.6. Player selects an opponent piece to be removed
- 10a.7. Player removes the opponent’s piece off the board

10a.8. Player has removed one of three opponent’s piece off the board leaving two opponent pieces on the board; game ends
10a.9. Player composes message to the opponent player describing the location of the man that is to be removed
10a.10. Player sends message to the other player and Board saves state to the database
10a.11. The opponent player receives the message and they remove the selected piece as per the message
10a.12. The game continues

Use Case 3: Move

Use Case Name: Player moves their piece
Primary Actor: Player
Goal: Move Piece
Stakeholder List: Player 1 / Player 2
Initiating Event:-Players turn is initiated.

Main Success Scenario:

1. Player selects their man to move
2. Board highlights adjacent move locations for selected man
3. Player selects one of the highlighted spaces to move their man
4. Player's man moves to the selected space and Game continues
5. Player's board saves their board's state to the database.
6. The other player's board updates their board to follow the state in the database.
7. Game initiates the other player's turn

Post Conditions:
- Other player's turn begins

Alternate Flow / Extensions:

2a. Player only has 3 men left on the board
2a.1. Board highlights every empty intersection on the board
2a.2. Player selects one of the highlighted spaces to move their man
2a.3. Player's man moves(flys) to the selected space and Game continues
2a.4. Player's board saves their board's state to the database.
2a.5. The other player's board updates their board to follow the state in the database.
2a.6. Game initiates the other player's turn

4a. Player's man moves to the selected space and mill is created
4a.1. Player selects one of the opposing player's men to remove from the board
4a.2. The board removes the selected man from the game
4a.3. Player's board saves their board's state to the database.
4a.4. The other player's board updates their board to follow the state in the database.
4a.5. Game initiates the other player's turn

4b. Player's man leaves a mill, destroying it, then moves to the selected space
4b.1. The board sets the man to being removable from the board.

4b.2. The board sets the other former mill members to be removable from the board.
4b.3. Player's board saves their board's state to the database.
4b.4. The other player's board updates their board to follow the state in the database.
4b.5. Game initiates the other player's turn

Use Cases Utilized:

- Use Case 2: Remove a piece

Scenario Note:

- **Player 1:** White.
- **Player 2:** Black
- **Mill:** Three of a players pieces placed adjacent to one another
- **Fly:** Placing a piece on any free spot on the board

Use Case 4: End Game by removing a piece

Use Case Name: End game by removing a piece
Primary Actor: Player
Goal: To win the game
Stakeholders List:
Player 1 and Player 2
Initiating Event: Player 1 moves a piece
Main Success Scenario:

5. Player 1 creates a Mill
6. Player composes message to the opponent player describing the location of the move
7. Player sends message to the other player through the internet
8. The opponent player receives the message and they update their board as per the message
9. Player selects an opponent piece to be removed
10. Player removes the opponent's piece off the board
11. Player composes message to the opponent player describing the location of the man that is to be removed
12. Player sends message to the other player and the board saves the current state to the database
13. The opponent player fetches results from the database.
14. Game verifies that only 2 opponent pieces remain on the board
15. Game fetches game record, which is retrieved by both players via the internet declaring the winner and loser

Post Conditions: Game ends and Player 1 wins
Alternate flows or exceptions:
N/A
Use Cases Utilized:

- Use Case 2: Remove a piece
- Use Case 3: Create a Mill

Scenario Note:

- **Player 1:** Black
- **Player 2:** White
- **Mills:** Three of one player's pieces (men) lined up horizontally/vertically

Use Case 5: End Game with no valid moves

Use Case Name: End Game with no valid moves
Primary Actor: Player
Goal: To win the game by leaving opponent with no valid moves
Stakeholders List:
Player 1 and Player 2
Initiating Event: Player places a piece on a intersection
Main Success Scenario:

4. Board verifies move
5. Player composes message to the opponent player describing the location of the move
6. Player sends message to the other player through the internet and board state is updated in database
7. The opponent player updates their board to follow the state in the database.
8. Player 2 is blocked with no more valid moves available
9. Game verifies that no valid moves can be made
10. Game fetches game record, which is retrieved by both players via the internet declaring the winner and loser

Post Conditions: Game ends and Player wins the opponent has no valid places to move their pieces to
Alternate flows or exceptions:
1a. Player 1 has made an illegal move
    1a.1. Board verifies move
    1a.2. Board requests Player to do another move?
    1a.3. Player selects a valid man to move
2a. Player makes a perfect play
    2a.1. Game ends in draw with perfect play
Use cases Utilized:

- Use Case 5: move piece
- Use Case 3: create a Mill

Scenario Note:

- **Player 1:** Black
- **Player 2:** White
- **Mills:** Three of one player’s pieces (men) lined up horizontally/vertically

Application - Detailed List of Classes

Referee

Class Description:

The Referee class is meant to act as a judge or a referee in the game. This class has the single responsibility of this class is to verify if the game’s rules are being obeyed by the users. The Referee class communicates with the Game class, Board class and Man class in order to carry out it’s responsibility.

Instance Variables:

<private> black_mill_count - Holds count of mill for player playing black

<private> white_mill_count - Holds count of mill for player playing white

<public> white_can_fly: Boolean - This boolean variable holds the value true or false for if white pieces can fly on the board. If this variable holds true then white players can move their piece on any available space on the board (e.g. fly). If this variable holds false then the white players can't fly and must make a move that is in an open adjacent point on the board.

<public> black_can_fly: Boolean - This boolean variable holds the value true or false for if black pieces can fly on the board. If this variable holds true then black players can move their piece on any available space on the board (e.g. fly). If this variable holds false then the black players can't fly and must make a move that is in an open adjacent point on the board.

<**public> mills: Hash <mill_coordinates, owner: String>** - This variable shows a hash value of the number of mills created and the owner of the mill

<public> selected_man: man - this variable hold the man piece that the user selects;

<public> selected_intersection: intersection - this variable holds the intersection that is selected by the user.

<public> current_state: Array<Intersection> - holds current state of board

<public> current_player_color: enum {'white', 'black'} - this holds current player color

Constructors:

<public> initialize() - This is the constructor for the class Referee and is used when instantiating an object of type Referee.

Methods:

<public> verify_mill(intersectionArr: array<Intersection>, current: Intersection):Boolean - This method will take in the current state of the board and the intersection to which the player is moved. This method will check preset combinations of valid mill locations from the mill's instance variable with the current state of board and the intersection to which the player has moved to. If a mill is created then mill count is updated by adding 1 and men in the combination are set to non removable. If a mill is broken then mill count is updated by removing 1 and men in from the broken mill are set to removable. Finally, this method returns true if a mill was created and false if no mill was created.

<public> is_valid_selected_man(selected_man: Man) : Boolean - this method takes in the current selected man and checks to see if the color of the Man matches the color assigned to the current player. If colors do not match this returns false, else this returns true.

<public> can_fly(numOfMen: integer) : Boolean - This method will verify if a player can fly. This method will use the numOfMen to determine if the current player can fly. If the number of Men is equal to 3 then a true value will be returned, verifying that a player can fly.

<public> is_valid_move(from: Intersection, to: Intersection, board : Board) : Boolean - This method verifies if a potential move is valid or not. A valid move means that none of the rules for moving men is violated. An example of such a rule would be: a man can't move to a point that is already occupied. Here the method takes in a from intersection, to intersection and a board. With these parameters the method uses the current location (from) and the new location (to) and verifies with the board to make sure the move being made is legal. The is_valid_move method returns a boolean; if it returns true then that means that the potential move is valid and the player is free to move their piece there. Alternatively, if it returns false that means that the potential move is not valid and the player can't move to that location.

<public> is_valid_removal(man:String) - This method checks if the removal of an opponent player's man is valid or not. The reason for this method is that Nine Men's Morris has a few rules around removing an opponent player's piece. One of those rules is that if there are other opponent men available for removal then an opponent's man in a mill cannot be removed. This method checks that such rules are not violated. The method returns a boolean value; if it returns true that means the removal is valid. Alternatively, if it returns false, then that means that the removal is invalid.

<public> is_valid_removal(selected_Intersection: Intersection, to_remove: Man, board: Board) - This method checks if the removal of an opponent player's man is valid or not. The reason for this method is that Nine Men's Morris has a few rules around removing an opponent player's piece. One of those rules is that if there are other opponent men available for removal then an opponent's man in a mill cannot be removed. This method checks that such rules are not violated. This method takes in the selected intersection, the man to remove and the board to validate the removal. The method returns a boolean value; if it returns true that means the removal is valid. Alternatively, if it returns false, then that means that the removal is invalid.

<public> winner_check(board: Board) : enum('white', 'black') - this methods checks the number of

<public> determine_available_moves(intersection : Intersection, board : Board) : Array<Intersection> - This method is supposed to find out all the available moves currently on the board. It takes in an intersection and a board object. The intersection holds the current location of the man the player wants to move. The board object holds the current state of the board. The method returns an array of all the intersections which are available for the man to move to.

Game

Class Description:

The game class is supposed to represent the actual game being played. In other words it holds information about every game that is played, with each game being an instance of the Game class. An example of using this class would be if the user decided to play a new game and a new instance of the game class is created, the user and its opponent are assigned either white or black and that is noted in the instance of this game class, a referee is created of class Referee and that is also contained in the game class. Other information such as the winner and the current turn of the players are also stored. The methods help the transition between player turn, end of the game, which player goes first, etc. An instance of the Game class can also be thought of as a game session.

Instance Variables:

<private> board: Board - Holds an instance of the board where this game session will be played on

<private> white: Player - Holds the player which has the white men

<private> black: Player - Holds the player which has the black men

<private> winner:enum{'white', 'black'} - Holds the winner of this game session (determined at the end of the game)

<public> current_turn: colour:enum{'white', 'black'} - Holds which player's (black or white) turn it is.

<public> phase: int- The current game's phase, 1-setup, 2-main, 3-end

<public> randomize(name: string): Player - this method takes in the name of the host player from rails and assigns that user a random color.

Constructors:

<public> initialize() - This is the constructor for the Game class. It initializes an instance of the Game class including its instance variables. This method creates two player objects and assigns to the instance variables "white" and "black". It also creates a new board and sets that equal to the instance variable "board". phase will be set to 1

Methods:

<public> update_phase(): void - updates current game phase., just does phase = phase +1 to advance to the next phase

<public> order_player(): void - This method generates order of Player by randomly assigning a colour of either white or black and assigning it to a player. As per the rules, White always goes first, this means the player assigned white will take the first turn.

<public> switch_player_turn(): colour: enum{'white', 'black'} - This method toggles between each player's turn. This method returns an enum :white or :black depending on whose turn it is. It essentially lets the game session know which player's turn it is after a turn has already ended.

<public> end(): void - This method ends the game session. Once a winner is declared, then this method destroys the board object and the player object which consequently destroys the player's bag object as well. Finally, the Game instance ends this game session.

Board

Class Description:

This class contains the information for a game board. Specifically, the board class contains an array of all the intersection points on the board and also keeps track of how many pieces are currently on the board for both colors. The board allows for multiple actions which allow the players to use their game pieces (men) to interact with the game board and intersections.

Instance Variables:

<public> num_white_man: Integer - This instance variable hold the number of white pieces (man) remaining on the board

***<public>* num_black_man: Integer -** This instance variable hold the number of black pieces (man) remaining on the board

<public> ref: Referee - This variable holds an instance of the Referee class which helps ensure that no rules are violated.

<public> intersections: Array<Intersection> - This variable holds an array of Intersection objects. This represents all the intersection points on the board and its information.

<public> selected_man: Intersection - This Variable holds the intersection containing a man piece that is selected by the user.

<public> selected_intersection: Intersection - The variable holds the selected intersection that the user wishes to interact with.

Constructors:

<public> initialize() - This is the constructor for the Board class and is used when instantiating an object of type Board.

Methods:

<public> get_num_men(colour: enum{'white', 'black'}):Integer - This method will check the value of colour provided and will return the number of white/black men as an integer based on the colour

<public> set_intersection(intersection : Intersection): boolean - This method takes in an intersection and updates that intersection in the instance variable “intersections” which contains an array of all intersections on the board.

<public> place(man: Man, coordinates: string): boolean - This method is called during the setup phase while the player is taking men from the bag. This method takes in the Man pulled from the bag and coordinates to its placement as parameters. This method then updates the arraylist of intersections with the updated placement.

<public> select_intersection(coordinates: string): Intersection - This method will receive a coordinate and will select that intersection and return it.

<public> move(current: Intersection, next: Intersection): Boolean -
This method takes in a current intersection and a next intersection. The method will then send this information to the referee to verify if the piece from the current intersection is able to displace itself to the next intersection legally. Upon passing all the checks, the method will update the intersection array list instance variable by calling the set_intersection method. Based on the validity of the move, this method will return true of false

<public> get_selected_intersection(): Intersection - This method returns the selected intersection

<public> get_selected_man(): man - This method returns the selected man. This can be used to remove the selected man or use the information of the selected man to verify required calculations by the referee.

<public> update_board(new_state: Array<Intersection>**): Boolean** - This method takes in the state of board from player A and updates for player B allowing all players to have the same state of board.

<public> get_board(): Array<Intersection> - This method returns the state of board to be sent to the opposite player to maintain state of board.

<public> remove(intersection : Intersection):Boolean - This method will receive an Intersection to be removed from the board and will check with the Referee instance variable to determine if this can be done. Upon passing all the checks, the method will update the intersection array list instance by variable by removing the Man from the specified intersection.

<public> select_man(coordinate: string): Boolean - This method takes a coordinate and returns True if a man is present at the intersection. This method will take the coordinate and check against the array of intersections to determine if a man is present at the specified coordinate. After verifying the Referee class will be called to determine if an appropriate man is selected using the is_valid_selected_man method. After the check is complete, the selected_man instance variable is updated and a true value is returned signifying that a man is present at the specified location.

<public> get_intersection_Array(): Array<Intersections> - This method acts as a getter, returning the boards intersection array.

Player

Class Description:

The player class represents the actors also known as the players of the game. The player will have a title and their coloured pieces. The player will be responsible for grabbing pieces from the bag and is also in charge of moving and removing their men on the board.

Instance Variables:

<public> name: string - The variable holds the name of the player, player 1 or player 2

<public> colour:enum{'white', 'black'} - The variable holds the players men (pieces) colour, player 1 will have white men and player 2 will have black men

<private> bag: Bag - The variable holds a set of men (9 pieces) for both players

Methods:

<public> get_name(): String - The method will return the name of the players, player 1 and player 2

<public> generate_bag(): bag - The method associates a bag with a player. At the beginning of the game, two bags are generated for player 1 and player 2 which will contain 9 coloured men. Player 1 will have 9 white men in the bag and player 2 will have 9 black men in the bag.

Constructors:

<public> initialize(name: String, colour: enum{‘white’, ‘black’}, board:Board) - This is a constructor for the Player class. It will contain which is player 1 and player 2 and the colour of their men. Takes in a board variable which holds an instance of the Board class which contains the current state of the board.

Man

Class Description:

The Man class represents the pieces used by the game. The only available information in this class will be about a single piece and the properties it holds. The Player and Board will utilize the Man class in order to verify positions and play the game.

Instance Variables:

<private> xy_coordinates:String - This instance variable shows the ‘xy’ coordinate the piece (man) holds on the board

<private> removable: Boolean - The removable variable will be True when the current piece can be removed from the board and False when the selected piece cannot be removed from the board.

<public> colour:enum{'white', 'black'} - A piece (man) can either be white or black

Methods:

<public> is_removable(): Boolean - This method will return the removable instance variable to show whether the current piece (man) is removable or not

<public> set_removable(is_removable: Boolean): void - This method will set the removable instance variable based on the value provided

<public> current_location(): String - This method will return the xy_coordinate value as a string along with the colour of the piece.

<public> set_location(coordinates:String): void - This method will receive the coordinates as a string and will update the Man’s location. This method will return void

Constructors:

<public> initialize() - This is the constructor for the class Man and is used when instantiating an object of type Man.

Bag

Class Description:

The Bag class represents the player’s bag and each player will have their own bag which will contain a set of 9 Men (pieces). This allows for an organized way to keep track of the pieces.

Instance Variables:

<private> Men : Array<Men> - A variable that contains an array of 9 men which the player will place onto the board.

Methods:

<public> Bag(colour: enum{‘white’, ‘black’}) - The method of the class, will take in a colour as an argument and assign that color to the 9 men that are created and placed in it's men array.

<**public**> **take_from_bag(): Man** - Returns a man object from the men array every time a piece is taken from the bag.

<**public**> **is_empty(): Boolean** - Returns true if there are no men left in the men array since there are no more pieces in the bag, otherwise false is returned.

Constructors:

<public> initialize(colour: enum{'white', 'black'}) - The constructor for the Bag class will take a colour value which will be used to determine the colour of the men in the bag for the player.

Intersection

Class Description:

The Intersection Class represents all the available intersections on the board. The Man class will use the Intersection class to show all the men occupying positions on the board.

Instance Variables:

<private> man:Man - This instance variable represents a Man object which will be used to keep track of piece on a certain intersection
<private> coordinates: String - The coordinates variable represents the position within the board the intersection will occupy.

Methods:

<public> set_man(man: Man): Boolean - This method takes in a Man object and sets that as the man on that intersection point by setting it equal to its instance variable man.
<public> get_coordinates(): String - This method will use the instance variable to return the coordinate. (This method acts as a getter for the coordinates)

Constructors:

<public> initialize(coordinates: String) - This is the constructor for the Intersection class. Upon initialization the coordinate will be set which will show the position the intersection will occupy on the board.

Rails - Data Model

Keys Legend:

- **Primary Keys**
- Foreign Keys

Player Database Table

Player (**playerId**, firstName, lastName, numOfWins)
- Ex. (**101**, John, Smith, 3)

playerID	**Primary Key**, Integer, Not Null
firstName	Varchar
lastName	Varchar
numOfWins	Integer

Game Database Table

Game (**gameId, playerId**, playerColour, winStatus)
- Ex. (**01, 101**, white, Won)
- Ex. (**01, 102**, black, Lost)

Note: winStatus can have one of the following values:
- Won

- Lost

gameId	**Primary Key**, Integer, Not Null
playerId	**Primary Key**, **Foreign Key**, Integer, Not Null
playerColour	Varchar
winStatus	Varchar

Man Database Table

Man (**gameId, pieceId**, playerId, position, colour, pieceStatus)

- Ex. (**01, W1**, 101, A3, white, on board)
- Ex. (**01, W2**, 101, NULL, white, in bag)
- Ex. (**01, B1**, 102, NULL, black, removed)

Note: Piece status can have one of the following values:

- On board
- Removed
- In Bag

gameID	**Primary Key**, **Foreign Key**, Integer, Not Null
pieceId	**Primary Key**, Varchar, Not Null
playerId	**Foreign Key**, Integer
position	Integer
colour	Varchar
pieceStatus	Varchar

Board Database Table

Board(**gameId**, **turnId**, playerId, stateOfIntersections)

- Ex. (**01, 09,** 101, {“A1”:””, “A4”:”W”, “A7”:”B”, “B2”:”B”, “B4”:””, … ,“G7”:”W”})

Note: This table stores the gameId, the id of each turn and the id of the player whose turn it currently is. Along with that, the table stores a json string containing all the intersection points on the board. If a point is empty then it is stored as an empty string, if the point is occupied then it stores either a B (black piece) or a W (white piece).

gameId	**Primary Key**, **Foreign Key,** Integer, Not Null
turnId	**Primary Key**, Integer, Not Null
playerId	**Foreign Key**, Integer
stateOfIntersections	Varchar

Rails - Controller Class Diagram Details

GamesController

Class Description:

Takes the user input that occurred in view and uses it with the appropriate method in the game model. Will also take what is outputted by the game model and convert it to a format that the view can render.

Instance Variables:

@current_game: Game - This variable holds the game model
@board:Board - This variable holds the board model
@referee:Referee - This variable hold the Referee model
@player_black:Player - This holds player black model
@player_white:Player - This holds player white model
@place_coordinates:String - This holds coordinates for placing a men during set up phase of the game
@from_coordinates:String - This holds coordinates of the current selected piece
@to_coordinates:String - This holds coordinates of the next selected position
@remove_coordinates:String: this holds coordinates of the selected to be removed coordinates.
@intersections:Hash - This will contain a key value pair where the key is the intersection, and the value is what man is on the intersection, "W" will mean white and "B" will mean black, "" will mean it is empty
@winner_colour - This will hold the colour of the game winner

Methods:

<public> index Will show the game browser for all available games, Users will be able to join a game in progress by using the game id, or create a new game lobby.

- This index method will only set a instance variable @games to equal Games.all (gets all the Game records)

The corresponding index.html will contain two links, the first link is for creating a new game which will be displayed on the screen using ruby's "link_to" function and the url used will be the path is new_game_path that is routed to games#new.

The second link will be for joining an existing game, the path is join_lobby_path and routes to games#join_lobby.

<public> new Create a new game record without saving it

<public> show Will have the following params passed: gameId and playerId, Will need these params to find the correct game record using Game.find_by(). An instance variable @game will be assigned the found game.

- check if player colour is nil on the associated game record, if it is then call setup_game
- get the stateOfIntersections from the associated board record (one that has gameId equal to this game's gameId), parse this JSON to a ruby hash object and set @intersections to equal this
- if session[:phase] is 1 or 2, render 'play'
- if session[:phase] is 3, render 'end'

<public> create This method will be used by game's new.html.erb view. Creates a new game object and player, this player will be the host. The url is create_game_path, routed to games#new and the method used is POST. The passed params will be :gameId, :firstName, and :lastName

- see if a record already exists that has the same :gameId from the passed params, if it does redirect to /new_game
- create a new game model object (@game)
- create and save a new Player model object with the passed params :firstName, and :lastName
- set the game record's gameId to be :gameId from the passed params
- set the game record's playerId to equal the newly created player.id
- set the session values :playerId and :gameId with the above values
- set the session :phase value to be equal to "0" (make sure it is a string, when its an int there are issues)
- set session :hostId value to equal to :playerId
- set the session [:pending_removal] to equal "false"
- save the new game record
- create and save a new board record, with the record's gameId set to :gameId from the passed params
- redirect to @game

<public> join_game
This method will be used by the game's join_lobby.html.erb view. Creates a new game object and player. This method is accessed from a form, the form details: The url is join_game_path, routed to games#join_game and the method used is POST. The passed params will include :gameId, :firstName, and last :lastName

- check if the game exists by looking for the game record with the passed gameId, if the record does not exist redirect to the main page (/index), otherwise:
- find the existing game record already created by the using the passed :gameId parameter, name it as @existing_game
- set the session[:hostId] to be the player id stored with in the @existing_game record
- create and save a new player record (@player) using Player.create() and pass :firstName and :lastName as arguments (these are included in the passed form params)
- create a new game record, @game
- set the new gameId to be equal to params[:gameId]
- set @game's playerId to be equal to the id of the new player record @player
- save the new @game record
- set the session[:playerId] to equal @player's id
- set the session [:gameId] to equal params[:gameId]
- set the session [:pending_removal] to equal "false"
- set the session [:phase] to equal "0"
- redirect to @game

<public> quit Allows a player to forfeit and quit the game, the remaining player is declared the winner. The game will then redirect the remaining player back to the browser

- The record for the game is updated for both users, the record for the player that quit is set as a loss
- Set the record for the other player to be a win
- Update session[:phase] for both players to be 4
- Redirect both players to the end game screen

<public> play The main way for a user to perform a game action, depending on the situation a specific form will be displayed on the screen and the form's parameters will describe what type of move was submitted (stored under the key: action), along with any coordinates. For example, the parameters for placement would have action equal to "place", if it was a removal, action would be set to "remove", and if it was a move, action would be set to "move". There would be if-else statements to see if the game's phase is still in the setup phase. If the game is still in the setup phase, then the setup_board helper function will be called. If the phase is not in the "setup" phase, then the params[: action] will be checked to see if it was a move or remove and the appropriate move or remove function will be called. At the end of the method, redirect_to @game will be called

- get the stateOfIntersections from the associated board record (one that has gameId equal to this game's gameId), parse this JSON to a ruby hash object and set @intersections to equal this
    - This is used to show whats on the gameboard
- check the session[:phase] if it is equal to "1"
    - call place(place_coordinates), with place_coordinates being the value passed from the form that POSTed to play, the value will be params[:place_coordinates]
- check the session[:phase] if it is equal to "2" and session[:pending_removal] is true
    - call remove(intersection_to_remove:string) with intersection_to_remove being the value passed from the form that POSTed to play, the value will be params[:remove_coordinates]
- check the session[:phase] if it is equal to "2" and session[:pending_removal] is false
    - call move(from:string, to: string) with "from" and "to" being being the values passed from the form that POSTed to play, the value will be params[:from_coordinates] and params[:to_coordinates]
- get the board array from Board by doing get_intersections_Array and set @intersection_array to contain it
- call (referee object).is_winner, a winner colour will be returned
    - if "white" or "black" is returned, call end_game
- call end_turn

<private> setup_game

- find @game by querying for the game record that is associated with session["gameId"] and game["playerId"]
- instantiate a Game object, @current_game
- instantiate a Board object
- instantiate a Referee object
- set Board's Referee object to be the new instantiated Referee object
- only the host will do this on their end(if session["playerId"] is equal to the playerId stored in the game record whose id is equal to session["gameId"]) find and check the game record's colour column using session[gameId], if it is null, then call randomize with the players name (get player name from the player record using Find by sessions["playerId"]) to get a Player object
    - Check the player object's color and assign the player as the white or black Game object instance variable
    - update and save the colour in the Game record associated with session["playerId"] and session["gameId"]
    - instantiate another player object for the other color and get the name by querying the game record that is associated with session["gameId"] and whose colour is set to nil
    - update and save the new colour in the Game record associated with a nil colour and session["gameId"]
    - send action cable message to the other player to create their player objects

        - the other player will receive the message and call game controller function create_players to create the two player objects on their side
        - send json over action cable containing your player id to the other player
            - games channel will have an if statement handling json data with a key: other_player_id
                - set session[other_playerId] to equal other_player_id from the json
- if the player is white (check the associated game record with id and player_id), set session["my_turn"] = true
- redirect to @game

<private> create_players

- query the Player records with session["playerId"] to get the first and last name,
- query the Game records with session["playerId"] to get the colour,
- create a player object that will represent the current player with the name and colour,
- query the Game records with session["gameId"] and the opposite colour to get the other_player_id
- query the Player records with other_player_id to get the first and last name,
- create a player object that will represent the other player with the retrieved name and opposite colour
- if the player is white (check the associated game record with id and player_id), set session["my_turn"] = true

<private> place(place_coordinates) places a man onto the board,

- find @game by querying for the game record that is associated with session["gameId"] and game["playerId"]
- take_from_bag() from the instantiate Player object's Bag variable which will return a man object (man_to_place)
- call place(man_to_place, place_coordinates),
    - if false is returned, redirect to @game
- call update_board_record to update the game's board record
- call get_num_men on the board for both white and black, if both are 9, set session[:phase] to "2" and the Game object's phase to 2
- send an action cable json string containing the method name: = "place" and "parameters" = place_coordinates"
- other player will call place with the above details (their games_channel will have an if statement that will handle received JSON string data)
- send an action cable json string containing the method name: = "change_phase" and "parameters" = "2"
    - there will be a if branch in the games_channel received function, when the JSON key is method_name, the other user will change their session[:phase] to the value stored under the "parameter" key

<private> move(from:string, to: string) sends a message to Game object to perform move and end_turn is called after completion. Here the game object calls move(from, to) and passes the parameters provided by the client. Board then sends this information to the Referee to validate the move.

- find @game by querying for the game record that is associated with session["gameId"] and game["playerId"]
- will have params[:from_coordinates] and params[:to_coordinates] pass in from the from that was passed to play
- call select_man(from_coordinates) on board to select the man to move
    - if false is returned, redirect to @game
- call select_intersection(from_coordinates) on board to select the intersection to move to
- call move(board.selected_man, board.selected_intersection)
    - if false is returned, redirect to @game
- update_board_record() with the intersection array from @board.get_intersections_Array()
- send an action cable json string containing the method name: = "move", "from" = from_coordinates, "to" = to_coordinates
    - other player will call move with the above details (their games_channel will have an if statement that will handle received JSON string data)
- verify_mill with Board.selected_intersection and @board.get_intersections_Array()
    - if true is returned and @board.playerId (whos current turn it is) equals session["playerId"], set session["pending_removal"] = "true"
    - redirect to @game (this will reload the page to show the move but it will still be the player's turn, although the form will change to show the form for removals, due to session["pending_removal"] = "true" have in html ruby to show this form and not show the move if session["pending_removal"] = "true")

<private> remove(intersection_to_remove:string) sends a message to the Game object with the selected intersection that the user wishes to remove men from. Game passes the parameter to remove(to_remove) and the Referee validates removal. intersection_to_remove will be the coordinates passed from the form in play, stored in params[:remove_coordinates]

- find @game by querying for the game record that is associated with session["gameId"] and game["playerId"]
- call Board.select_intersection(intersection_to_remove) to get an intersection @sel_intersection
- call Board.remove(@sel_intersection)
    - if false is returned, redirect to @game
- update_board_record() with the intersection array from @board.get_intersections_Array()
- send an action cable json string containing the method name: = "remove", key:"remove_coordinates" , value: params[:remove_coordinates]

    - other player will call remove with the above details (their games_channel will have an if statement that will handle received JSON string data)
- set session["pending_removal"] = "false"

<private> end_turn sends a message to Game object to call switch_player_turn() which updates the view so that only the current User whose turn it is will see controls on their screen, the other player will only see the board and an indication that it is the other player's turn.

- call @current_game.switch_player_turn()
- call update_board_record(array) with the array returned with @board.get_intersections_Array()
- update the board record that is associated with this game, (query for the boardId that has this game's game_id) to be the other player id (get this from session[other_player_id])
- broadcast action cable message to other player, a string: "player ended turn"
    - other player have a if branch in their data received function in games_channel, if data string "player ended turn", set session[:my_turn] = true
- set session[:my_turn] = false

<private> update_board_record(IntersectionArray) Updates the stateOfIntersections board record column to reflect what has been set in the board object
- find the board record for this game by querying for a board record that has a gameId equal to this game's gameId
- declare a variable @board_hash is a ruby Hash where each key is a moveable intersection on the board ex."A1", and its values are all blank ("")
- get the array of intersections using @board.get_intersections_Array() and store it in @intersection_array
- iterate through each element in @intersection_array
    - for an element, do element.get_man().color to get the colour, @man_colour
    - for an element, do element.get_coordinates(), @coords
    - then do board_hash[@coords] = "W" if @man_colour is white
        - else board_hash[@coords] = "B"
- save board record changes

<private> end_game The Referee validates game state to check if a player has won. Once all the validations are complete, the game ends with the winner being declared. The session[:phase] of the game is updated to 4.

- call (referee object).is_winner, a winner colour will be returned @winner_colour
    - update the game record that has the same gameId and player colour as @winner_colour so that the win status is "Won"
- update the game record that has the same gameId and opposite player colour as @winner_colour so that the win status is "Lost"

- change session[:phase] to "3"
- send a action cable message to the other user
- render 'end'

Rails - List of URL Calls

get '/browser', to: 'games#index'
get '/new_game', to: 'games#new'
get '/join_lobby', to: 'games#join_lobby'
get '/about', to: 'staticpages#about'
get '/play', to: 'games#play'

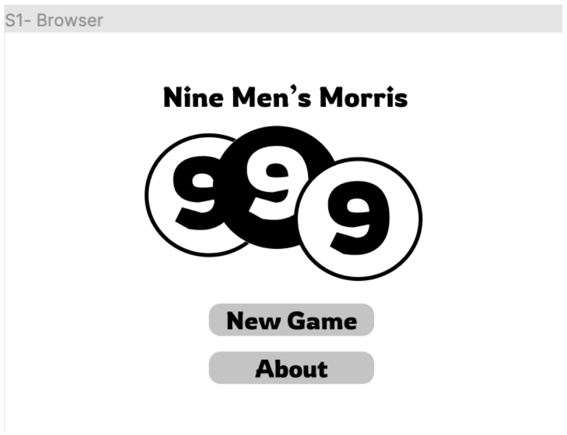
post '/join_game', to: 'games#join_game'
post '/play', to: 'games#play'
post '/quit', to: 'games#quit
post '/create_game', to: 'games#create'

resources :users
resources :games
resources :boards
resources :man

Rails - Web Pages

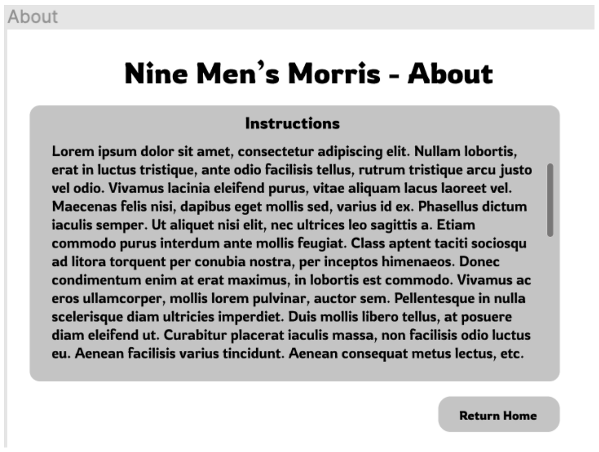
S1 is the homepage. It is the destination page/screen for any “Quit” buttons or any “Return Home” buttons. This is also the first screen the player sees when they attempt to play this game.

Home Page: GET /browser
Quit Button (before game has started): GET /browser
Return Home: GET /browser



The 2 buttons on the S1 page lead to different screens. The "New Game" button takes the player to the S2 - new game page. The "About" button takes the player to the about page which contains all the information and rules about the game. The about page has a button called "Return Home" which takes the player back to the browser home page.

About: GET/ about
Return Home: GET/browser

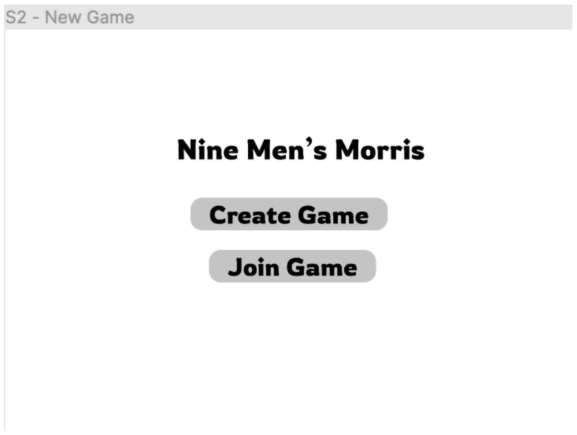


Nine Men's Morris - About

Instructions

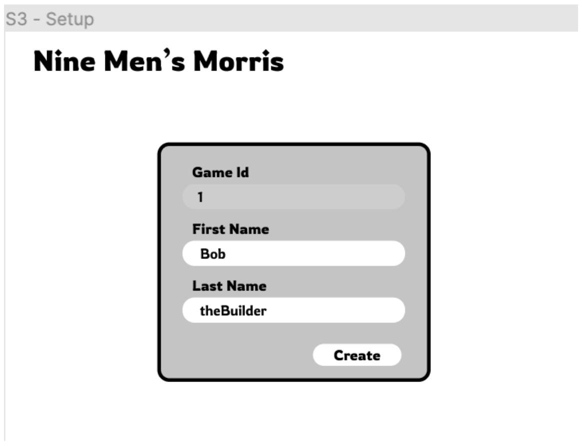
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam lobortis, erat in luctus tristique, ante odio facilisis tellus, rutrum tristique arcu justo vel odio. Vivamus lacinia eleifend purus, vitae aliquam lacus laoreet vel. Maecenas felis nisi, dapibus eget mollis sed, varius id ex. Phasellus dictum iaculis semper. Ut aliquet nisi elit, nec ultrices leo sagittis a. Etiam commodo purus interdum ante mollis feugiat. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec condimentum enim at erat maximus, in lobortis est commodo. Vivamus ac eros ullamcorper, mollis lorem pulvinar, auctor sem. Pellentesque in nulla scelerisque diam ultricies imperdiet. Duis mollis libero tellus, at posuere diam eleifend ut. Curabitur placerat iaculis massa, non facilisis odio luctus eu. Aenean facilisis varius tincidunt. Aenean consequat metus lectus, etc.

New Game: GET/new_game

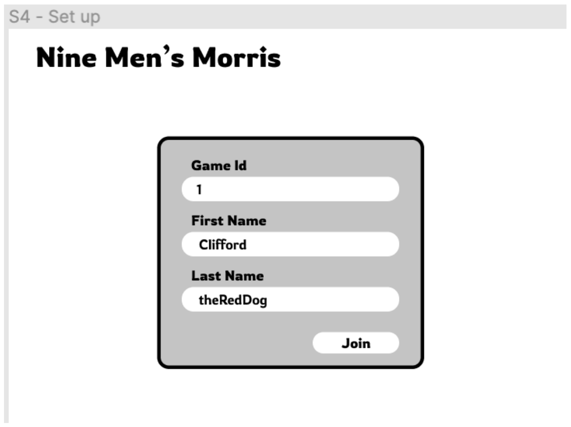


The S2 - new game page has two buttons. “Create Game” takes the player to the setup page where they can fill in their information and get a unique game id (S3). The “Join Game” page takes the player to the setup page and similar to before they can fill in their information, except this time (since they are the second player) they have to input the unique game id as well (S4).

S3 - Setup (Create Game) : POST /games
S4- Setup (Join Game): POST /Lobby



Game Id
1
First Name
Bob
Last Name
theBuilder


S4 - Set up

Nine Men's Morris

Game Id: 1

First Name: Clifford

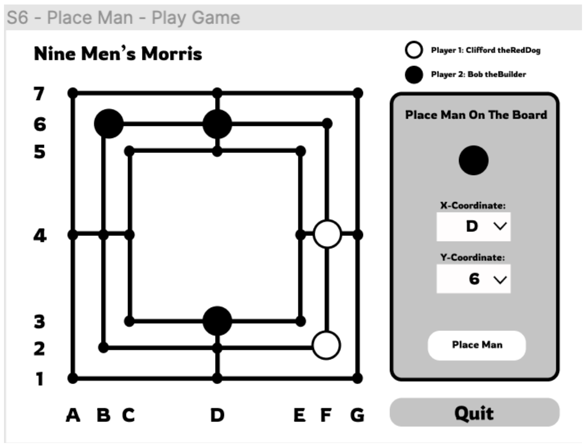
Last Name: theRedDog

The "Create" button on S3 and "Join" button on S4 will both take the player to S6.

S3 - Create Button: GET /play
S4 - Join Button: GET/play

The play page contains the game board and the drop downs that allows the user to decide where to place a man, move a man and/or remove a man. The drop downs are automatically adjusted to only show the available moves that a player can make. Upon clicking the "Place Man" button, the page refreshes and takes the player back to the updated play page. The "Quit" button takes the player back to the browser home page (S1).

S6 - Place Man Button: POST /play
S6 - Quit button: POST /quit


S6 - Place Man - Play Game

Nine Men's Morris

○ Player 1: Clifford theRedDog

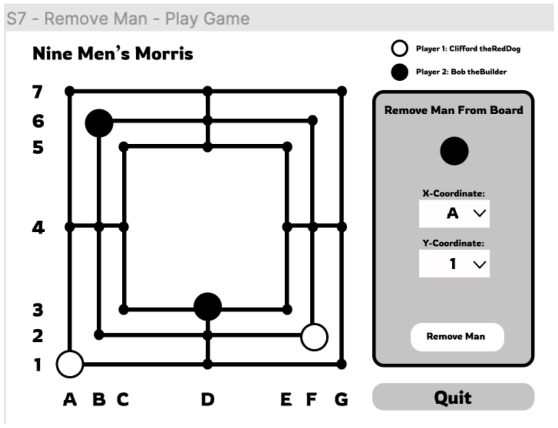
● Player 2: Bob theBuilder

Place Man On The Board

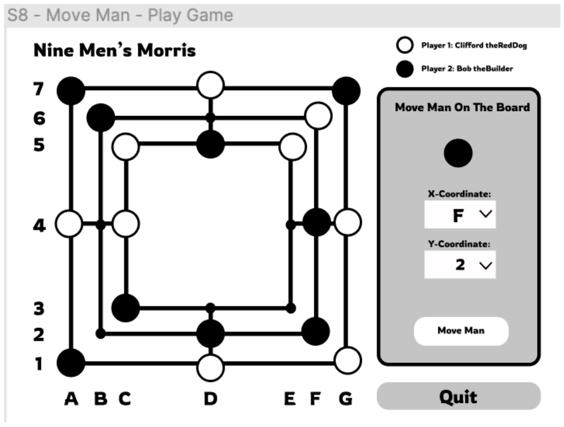
X-Coordinate: D

Y-Coordinate: 6

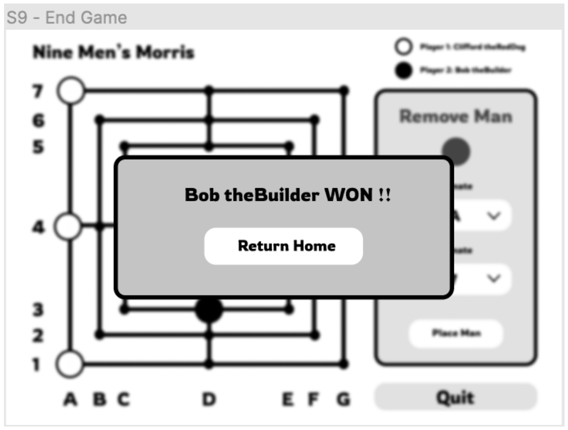
S7 - Remove Man Button: POST /play
S7 - Quit button: POST /quit



S8 - Move Man Button: POST /play
S8 - Quit button: POST /quit



After the game has finished and a winner has been declared, the Winner modal will be generated which will have a “Return Home” button that will take you to Browser page S1.
Return Home: GET /browser



Web Pages Design

The design of the webpages were done on Figma and can be achieved by adding the CSS and JS link tags to the header of the html file. This will allow you to take advantage of the bootstrap styling library and will help you to achieve the style presented in the figma diagram presented below. Both the link commands can be found on the bootstrap website.

https://getbootstrap.com/docs/5.1/getting-started/introduction/

However if you are still having difficulties adding bootstrap to your rails app, a resource which can help guide you is:

https://www.youtube.com/watch?v=EzCl-6etSGI

For our design we relied mostly on the Modal, Form and Button components within Bootstrap. All these components can be found using the search bar on the Bootstrap website.

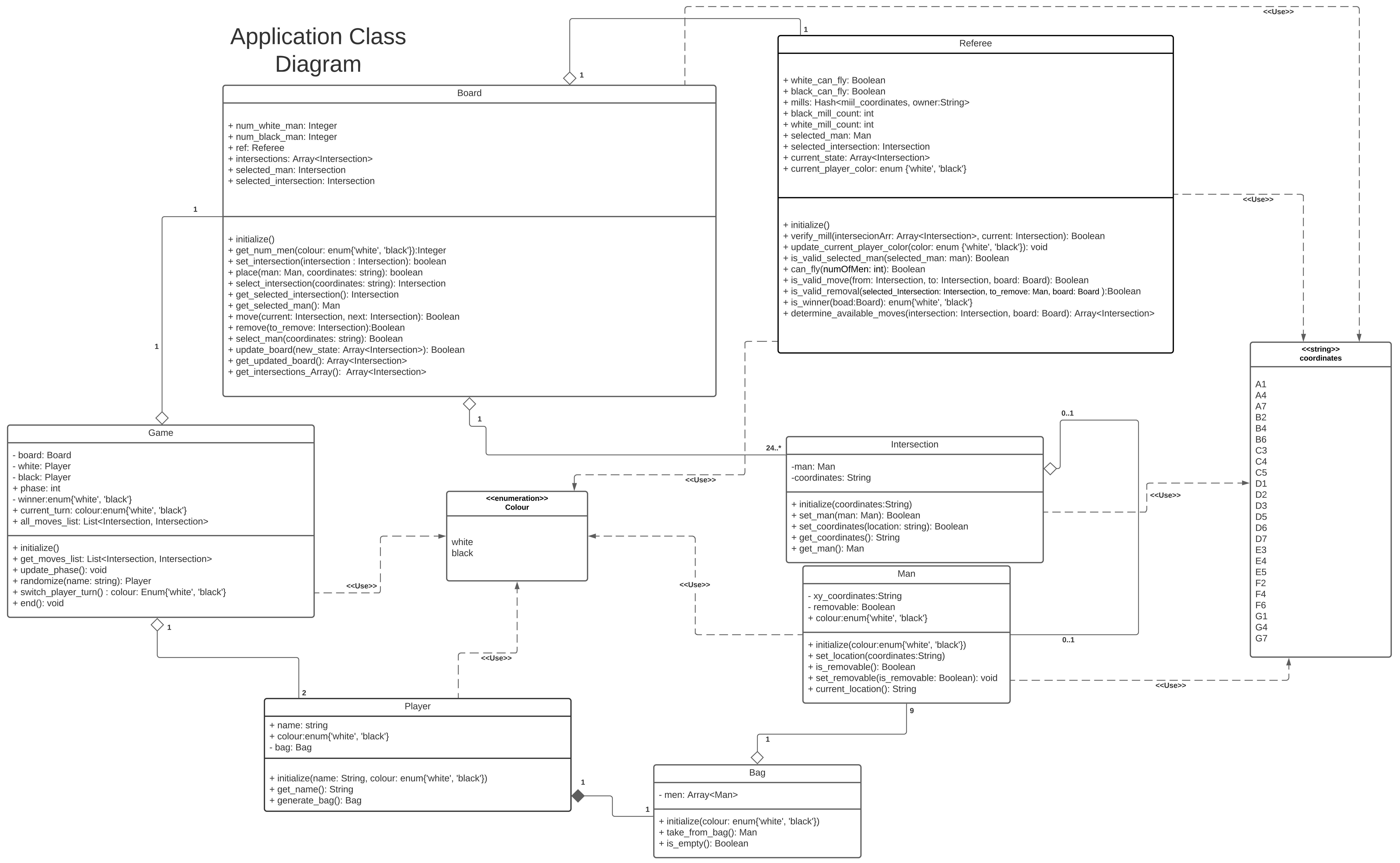
The html board representation could be a 27 * 27 cell table, where the intersections would be placed onto the correct cell, using ruby you can get a cell value from the @intersections from the games controller. the td cell would just have something like (<%=@intersections["7A"]%>) for the cell value.

DISTRIBUTION OF EFFORT

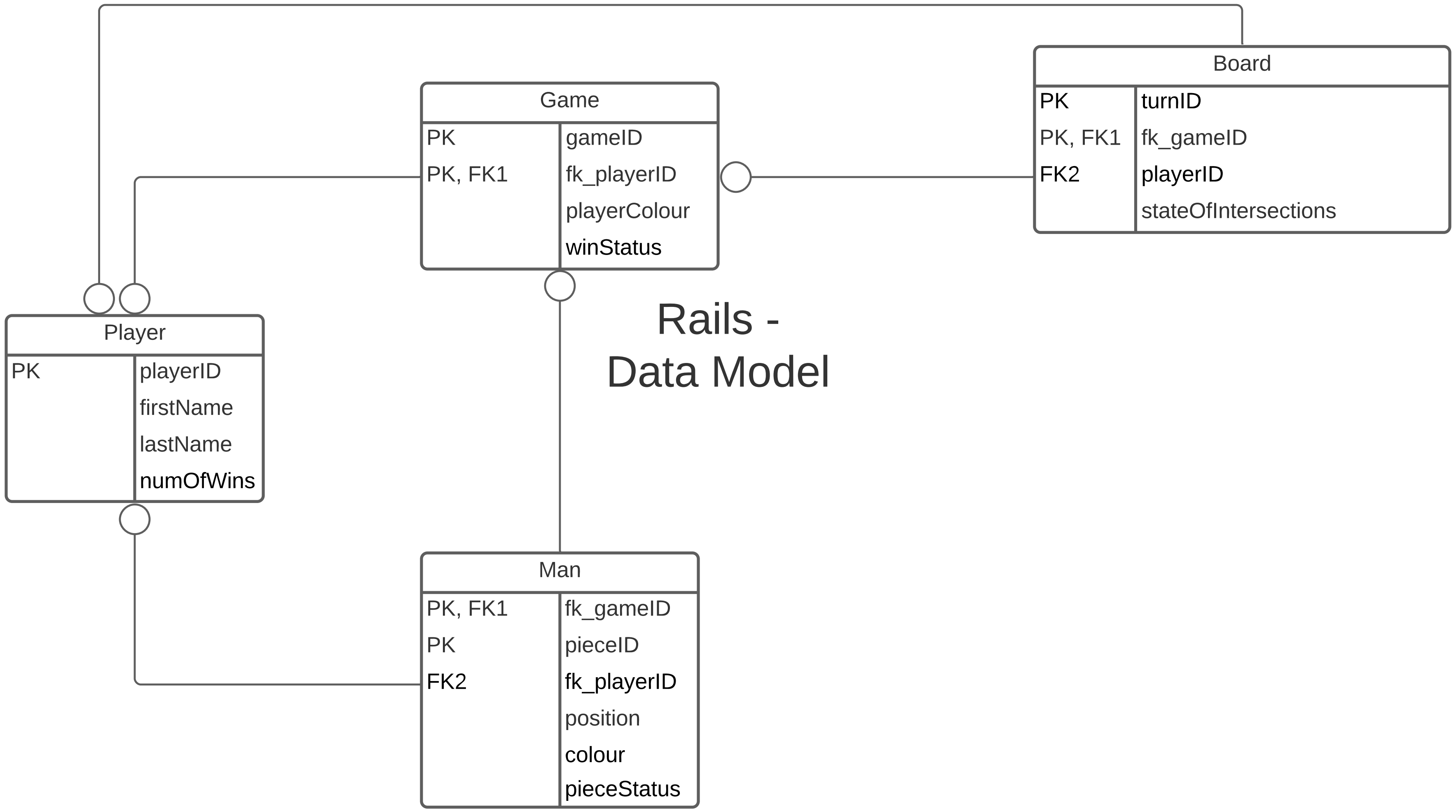
Dixant Patel: 1
Ryan Paul: 1
Nareshri Babu: 1
Jarryl Subang: 1
Siri Chandana Kathroju: 1

Application Class Diagram

Application Class Diagram

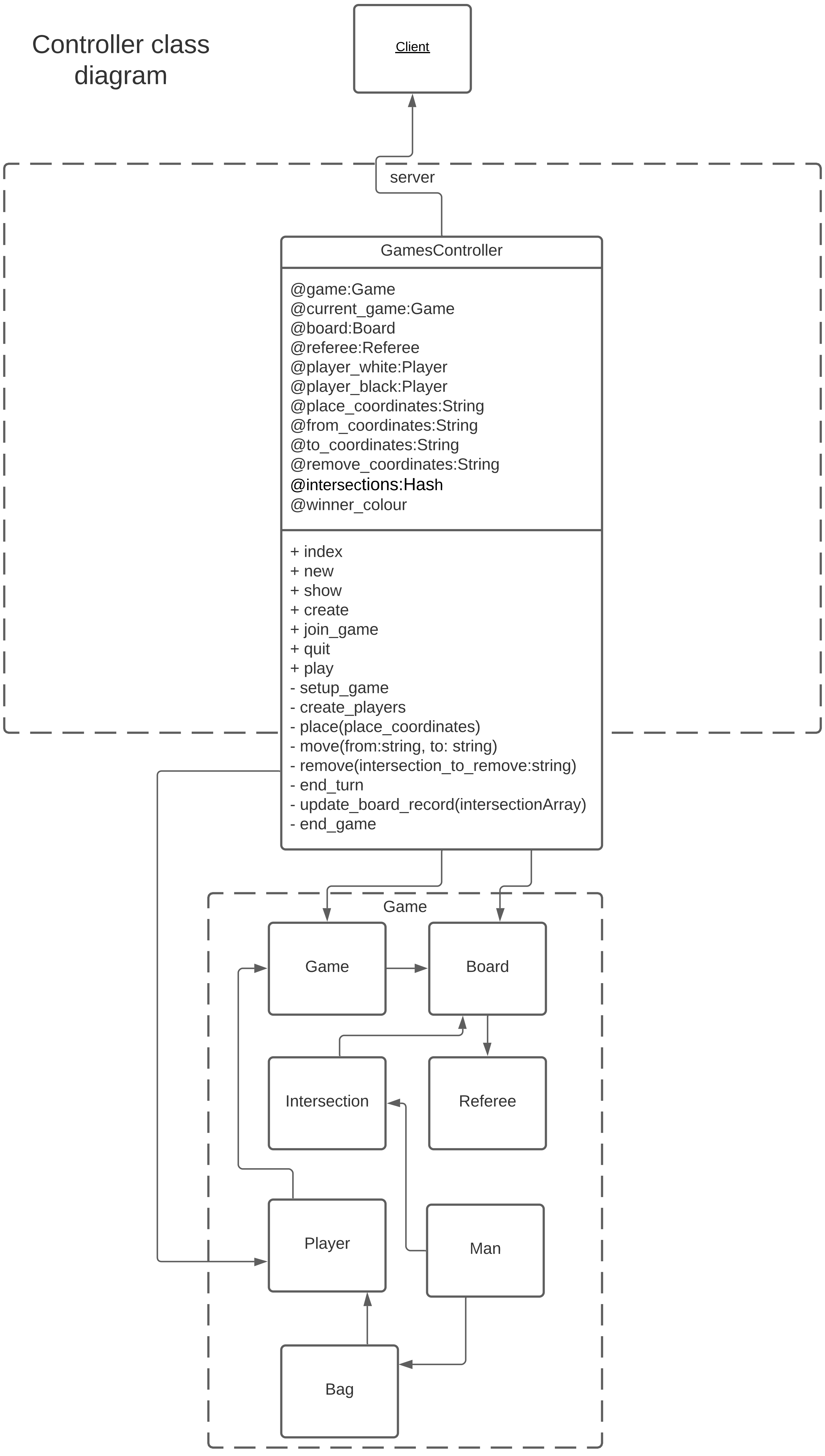

Board
+ num_white_man: Integer
+ num_black_man: Integer
+ ref: Referee
+ intersections: Array<Intersection>
+ selected_man: Intersection
+ selected_intersection: Intersection
+ initialize()
+ get_num_men(colour: enum{'white', 'black'}):Integer
+ set_intersection(intersection : Intersection): boolean
+ place(man: Man, coordinates: string): boolean
+ select_intersection(coordinates: string): Intersection
+ get_selected_intersection(): Intersection
+ get_selected_man(): Man
+ move(current: Intersection, next: Intersection): Boolean
+ remove(to_remove: Intersection):Boolean
+ select_man(coordinates: string): Boolean
+ update_board(new_state: Array<Intersection>): Boolean
+ get_updated_board(): Array<Intersection>
+ get_intersections_Array(): Array<Intersection>
Referee
+ white_can_fly: Boolean
+ black_can_fly: Boolean
+ mills: Hash<miil_coordinates, owner:String>
+ black_mill_count: int
+ white_mill_count: int
+ selected_man: Man
+ selected_intersection: Intersection
+ current_state: Array<Intersection>
+ current_player_color: enum {'white', 'black'}
+ initialize()
+ verify_mill(intersecionArr: Array<Intersection>, current: Intersection): Boolean
+ update_current_player_color(color: enum {'white', 'black'}): void
+ is_valid_selected_man(selected_man: man): Boolean
+ can_fly(numOfMen: int): Boolean
+ is_valid_move(from: Intersection, to: Intersection, board: Board): Boolean
+ is_valid_removal(selected_Intersection: Intersection, to_remove: Man, board: Board):Boolean
+ is_winner(boad:Board): enum{'white', 'black'}
+ determine_available_moves(intersection: Intersection, board: Board): Array<Intersection>
<<string>>
coordinates
A1
A4
A7
B2
B4
B6
C3
C4
C5
D1
D2
D3
D5
D6
D7
E3
E4
E5
F2
F4
F6
G1
G4
G7
Game
- board: Board
- white: Player
- black: Player
+ phase: int
- winner:enum{'white', 'black'}
+ current_turn: colour:enum{'white', 'black'}
+ all_moves_list: List<Intersection, Intersection>
+ initialize()
+ get_moves_list: List<Intersection, Intersection>
+ update_phase(): void
+ randomize(name: string): Player
+ switch_player_turn() : colour: Enum{'white', 'black'}
+ end(): void
<<enumeration>>
Colour
white
black
Intersection
-man: Man
-coordinates: String
+ initialize(coordinates:String)
+ set_man(man: Man): Boolean
+ set_coordinates(location: string): Boolean
+ get_coordinates(): String
+ get_man(): Man
Man
- xy_coordinates:String
- removable: Boolean
+ colour:enum{'white', 'black'}
+ initialize(colour:enum{'white', 'black'})
+ set_location(coordinates:String)
+ is_removable(): Boolean
+ set_removable(is_removable: Boolean): void
+ current_location(): String
Player
+ name: string
+ colour:enum{'white', 'black'}
- bag: Bag
+ initialize(name: String, colour: enum{'white', 'black'})
+ get_name(): String
+ generate_bag(): Bag
Bag
- men: Array<Man>
+ initialize(colour: enum{'white', 'black'})
+ take_from_bag(): Man
+ is_empty(): Boolean
<<Use>>
<<Use>>
<<Use>>
<<Use>>
<<Use>>
<<Use>>
<<Use>>
<<Use>>
1
1
1
1
1
24..*
0..1
0..1
1
2
9
1
1
1

Data Model


Rails -
Data Model
Game
PK gameID
PK, FK1 fk_playerID
playerColour
winStatus
Board
PK turnID
PK, FK1 fk_gameID
FK2 playerID
stateOfIntersections
Player
PK playerID
firstName
lastName
numOfWins
Man
PK, FK1 fk_gameID
PK pieceID
FK2 fk_playerID
position
colour
pieceStatus

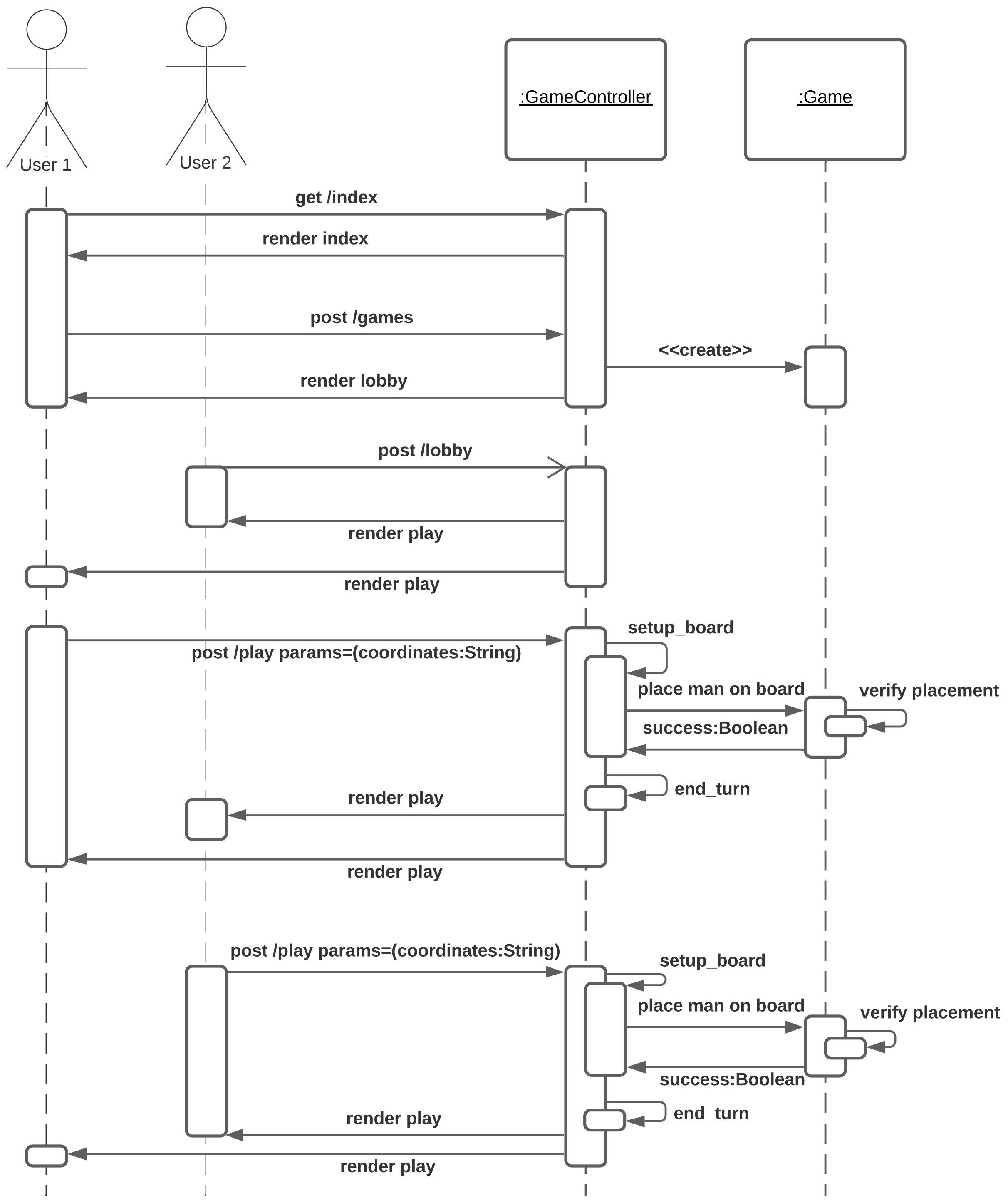
Controller Class Diagram

Controller class diagram

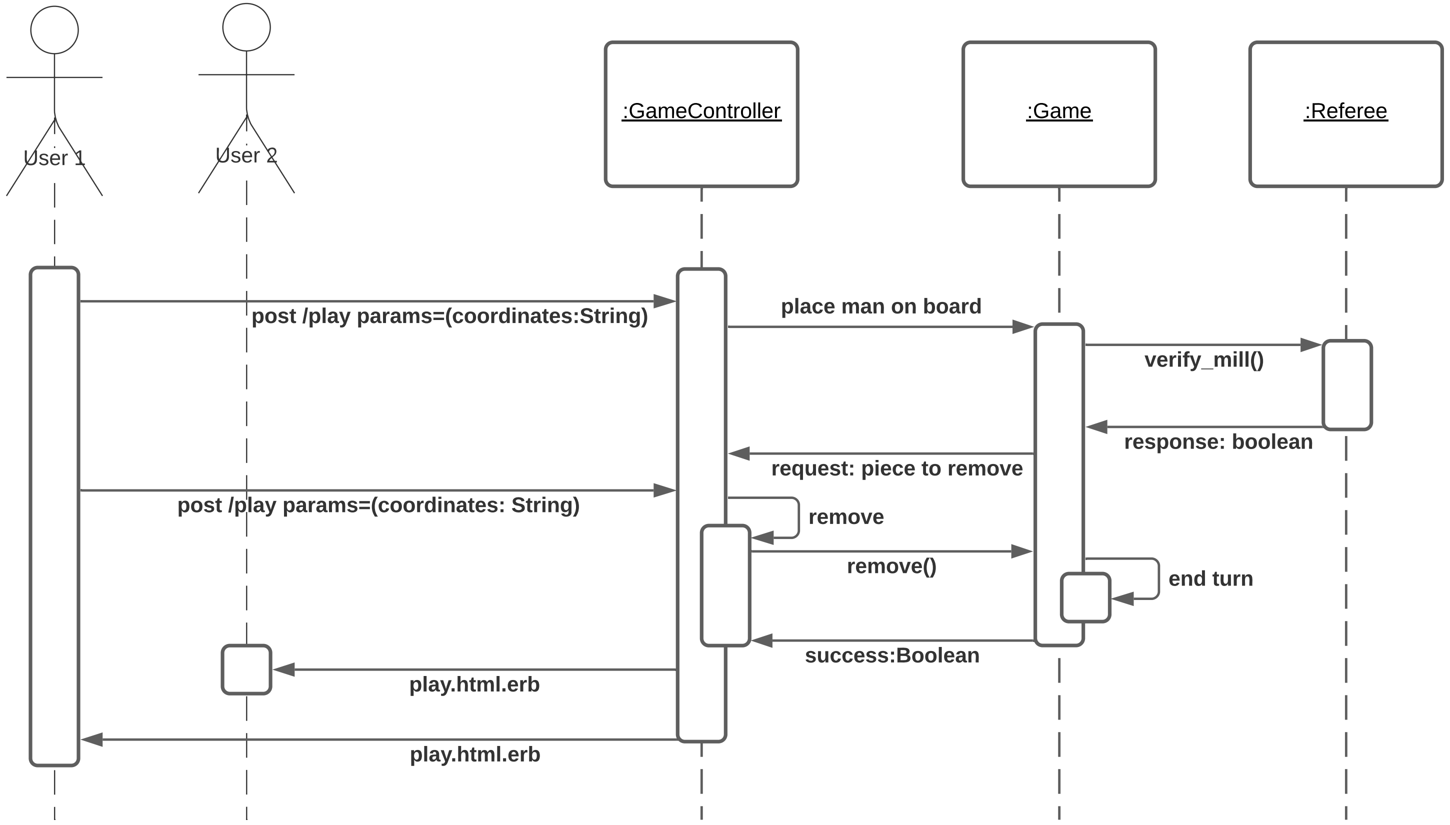


Client/Server Sequence Diagrams

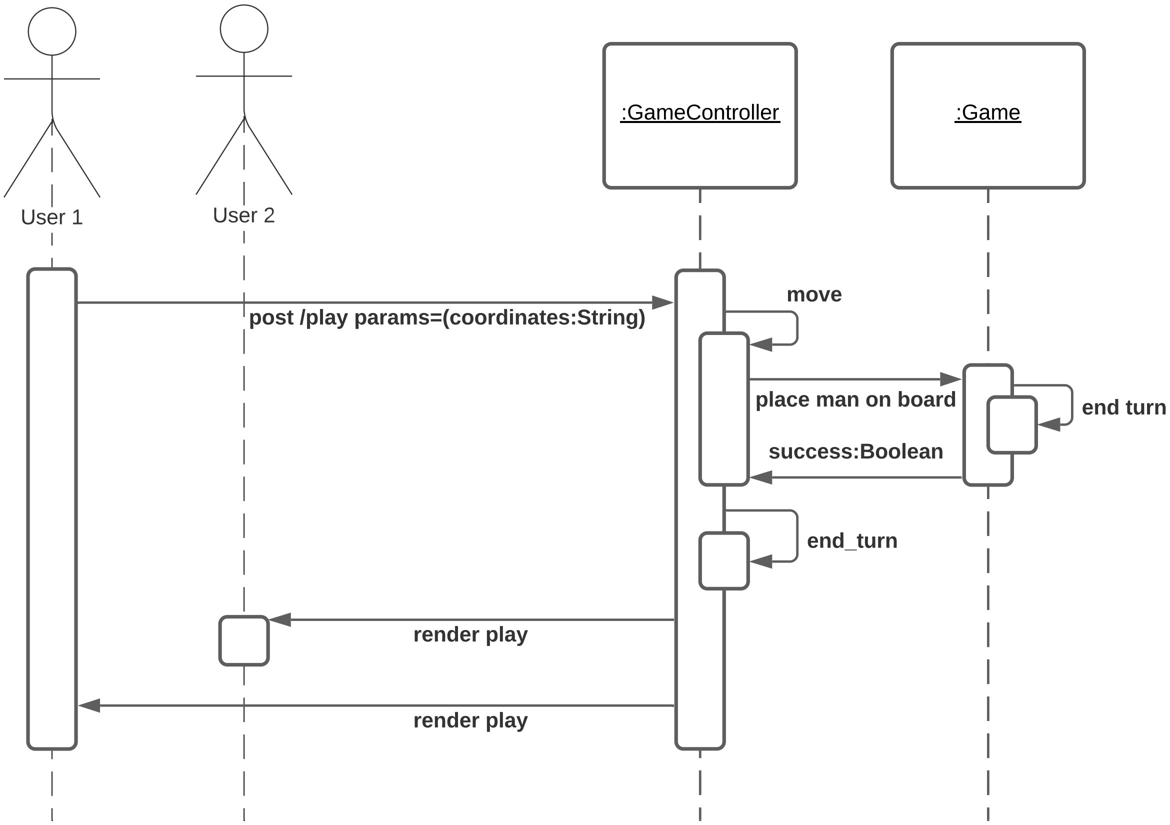
Use Case 1: setup Board



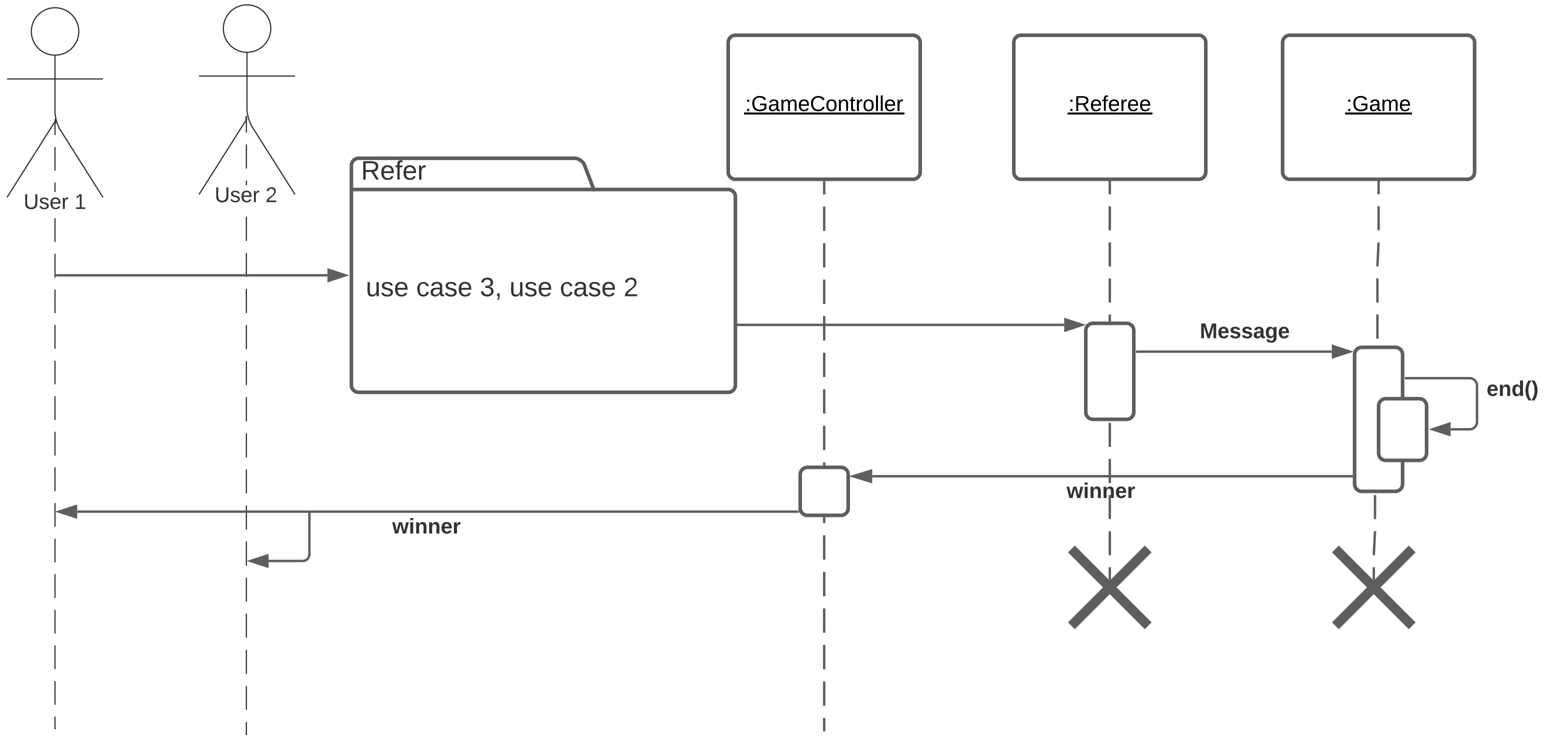
Use case 2: remove a piece



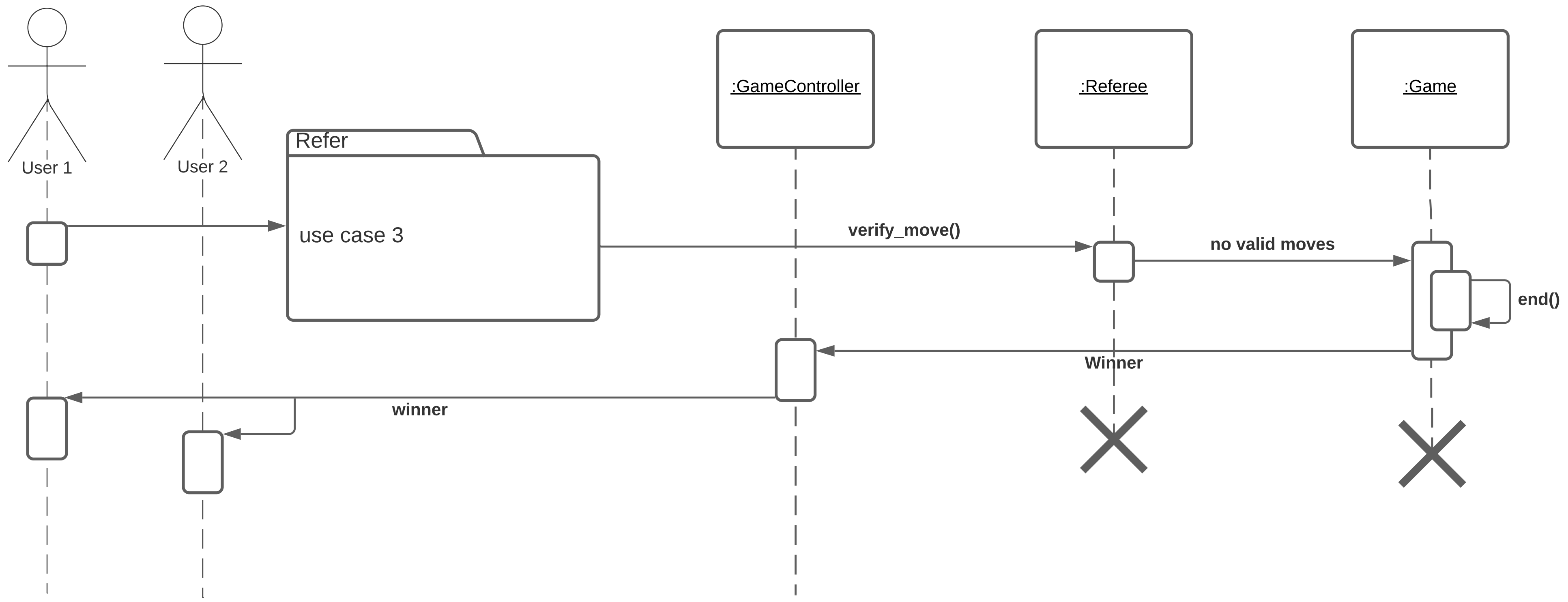
Use case 3: Move



Use case 4: End Game by removing a piece

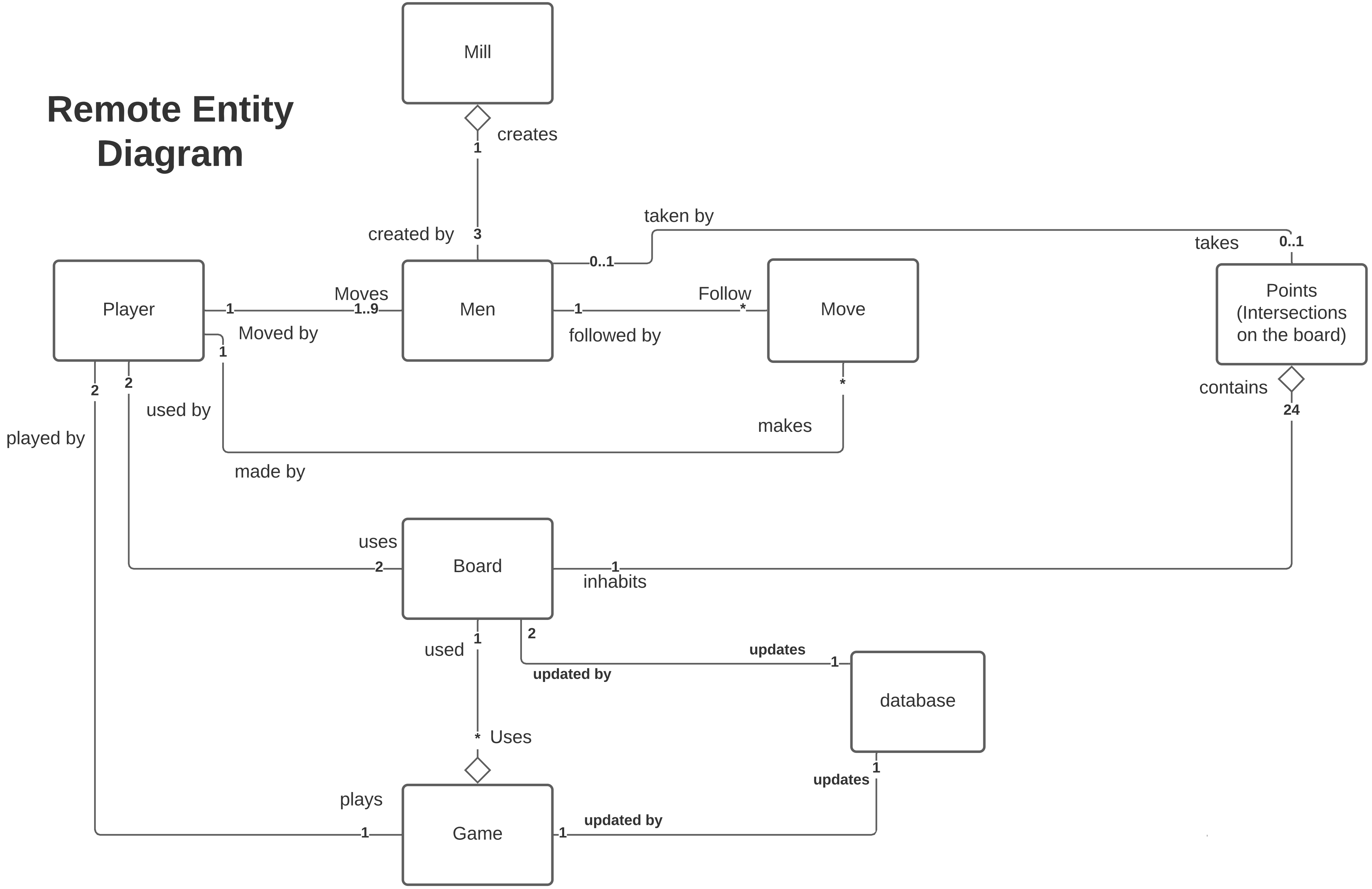


Use case 5: End Game with no valid moves



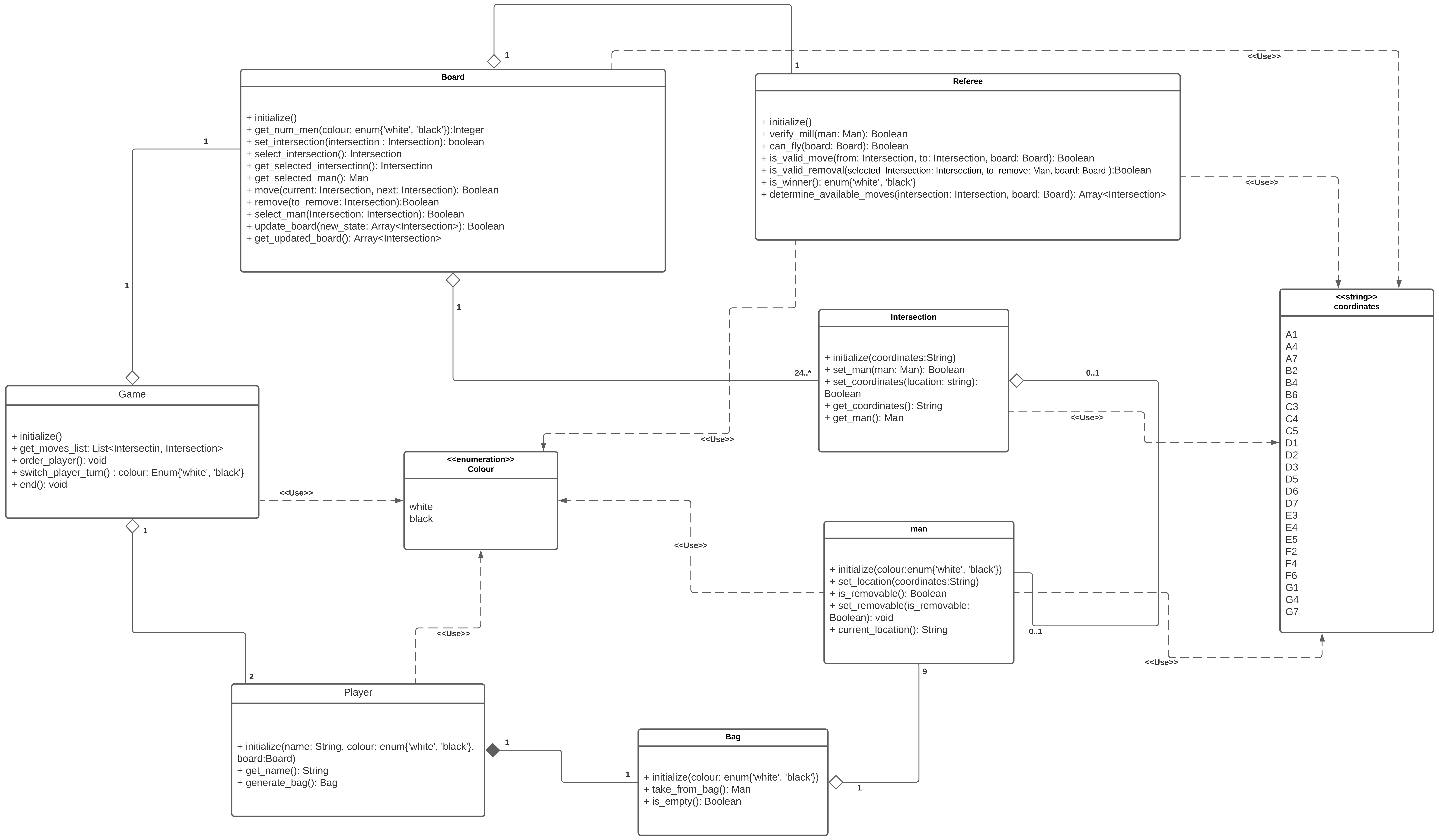
Entity Diagram

Remote Entity Diagram



Application Type Diagram

Application Type Diagram



Board

+ initialize()
+ get_num_men(colour: enum{'white', 'black'}):Integer
+ set_intersection(intersection : Intersection): boolean
+ select_intersection(): Intersection
+ get_selected_intersection(): Intersection
+ get_selected_man(): Man
+ move(current: Intersection, next: Intersection): Boolean
+ remove(to_remove: Intersection):Boolean
+ select_man(Intersection: Intersection): Boolean
+ update_board(new_state: Array<Intersection>): Boolean
+ get_updated_board(): Array<Intersection>

Referee

+ initialize()
+ verify_mill(man: Man): Boolean
+ can_fly(board: Board): Boolean
+ is_valid_move(from: Intersection, to: Intersection, board: Board): Boolean
+ is_valid_removal(selected_Intersection: Intersection, to_remove: Man, board: Board):Boolean
+ is_winner(): enum{'white', 'black'}
+ determine_available_moves(intersection: Intersection, board: Board): Array<Intersection>

<<string>>
coordinates

A1
A4
A7
B2
B4
B6
C3
C4
C5
D1
D2
D3
D5
D6
D7
E3
E4
E5
F2
F4
F6
G1
G4
G7

Intersection

+ initialize(coordinates:String)
+ set_man(man: Man): Boolean
+ set_coordinates(location: string): Boolean
+ get_coordinates(): String
+ get_man(): Man

Game

+ initialize()
+ get_moves_list: List<Intersectin, Intersection>
+ order_player(): void
+ switch_player_turn() : colour: Enum{'white', 'black'}
+ end(): void

<<enumeration>>
Colour

white
black

man

+ initialize(colour:enum{'white', 'black'})
+ set_location(coordinates:String)
+ is_removable(): Boolean
+ set_removable(is_removable: Boolean): void
+ current_location(): String

Player

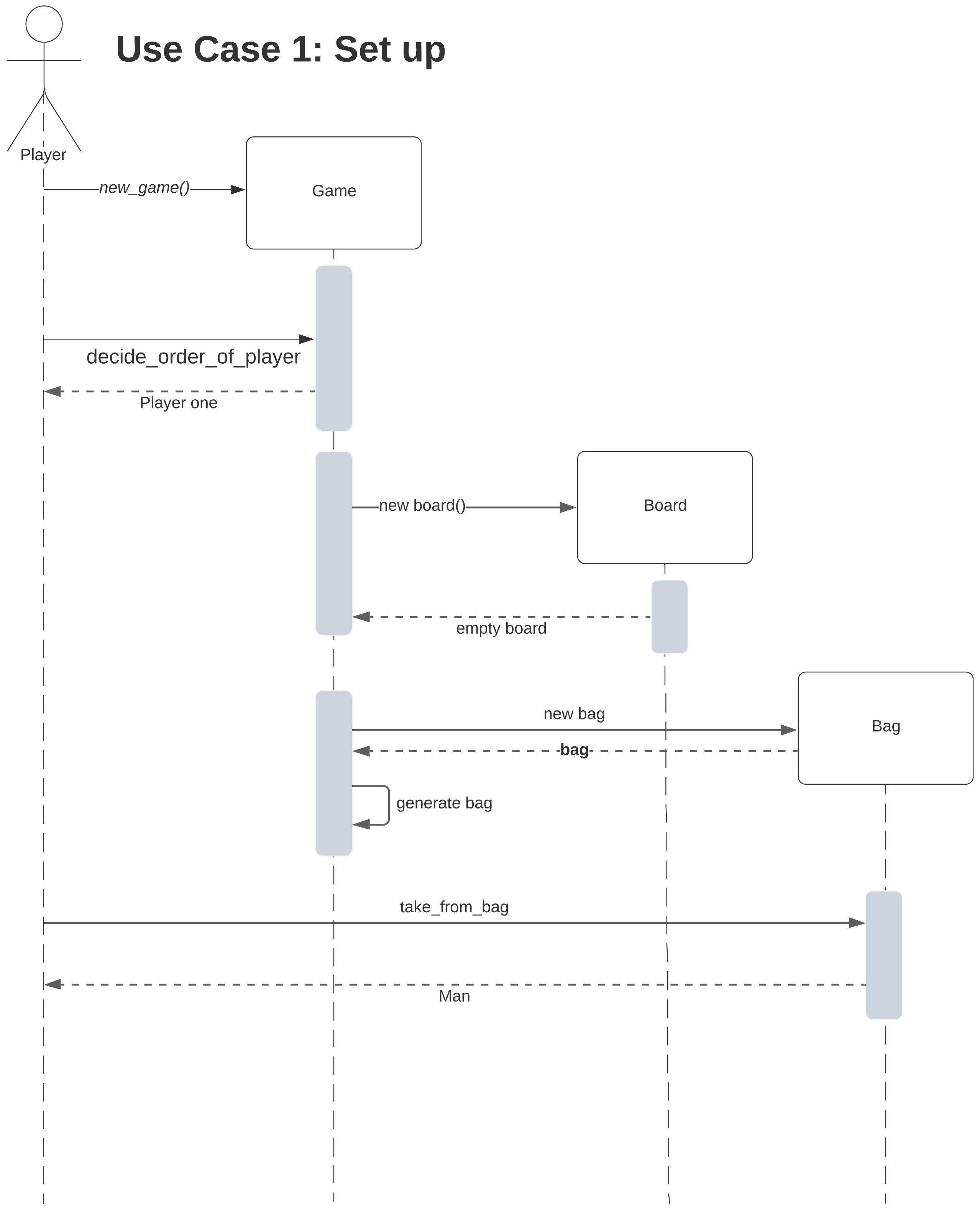
+ initialize(name: String, colour: enum{'white', 'black'}, board:Board)
+ get_name(): String
+ generate_bag(): Bag

Bag

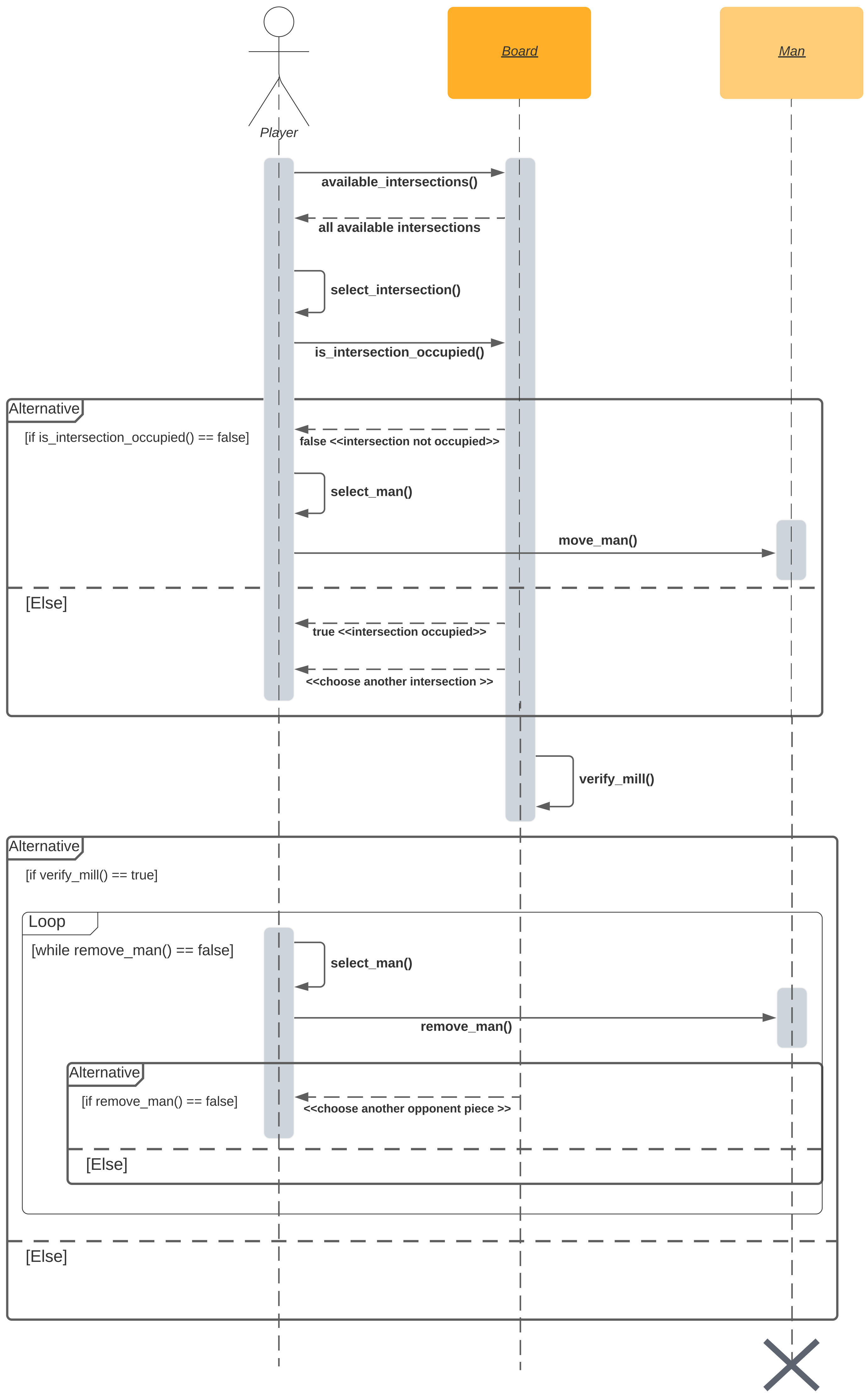
+ initialize(colour: enum{'white', 'black'})
+ take_from_bag(): Man
+ is_empty(): Boolean

Application Sequence Diagram

Use Case 1: Set up

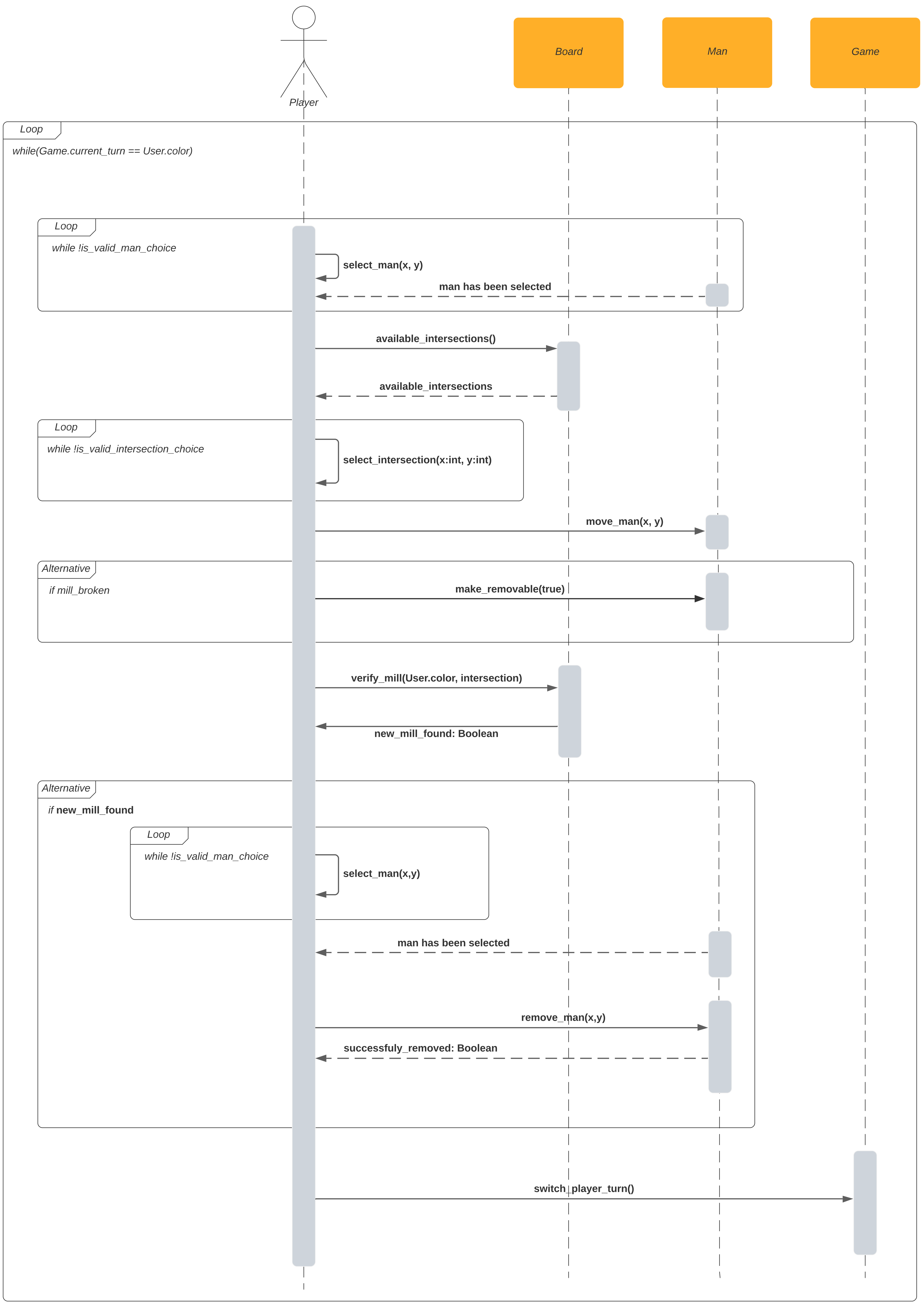


Use Case 2: Remove

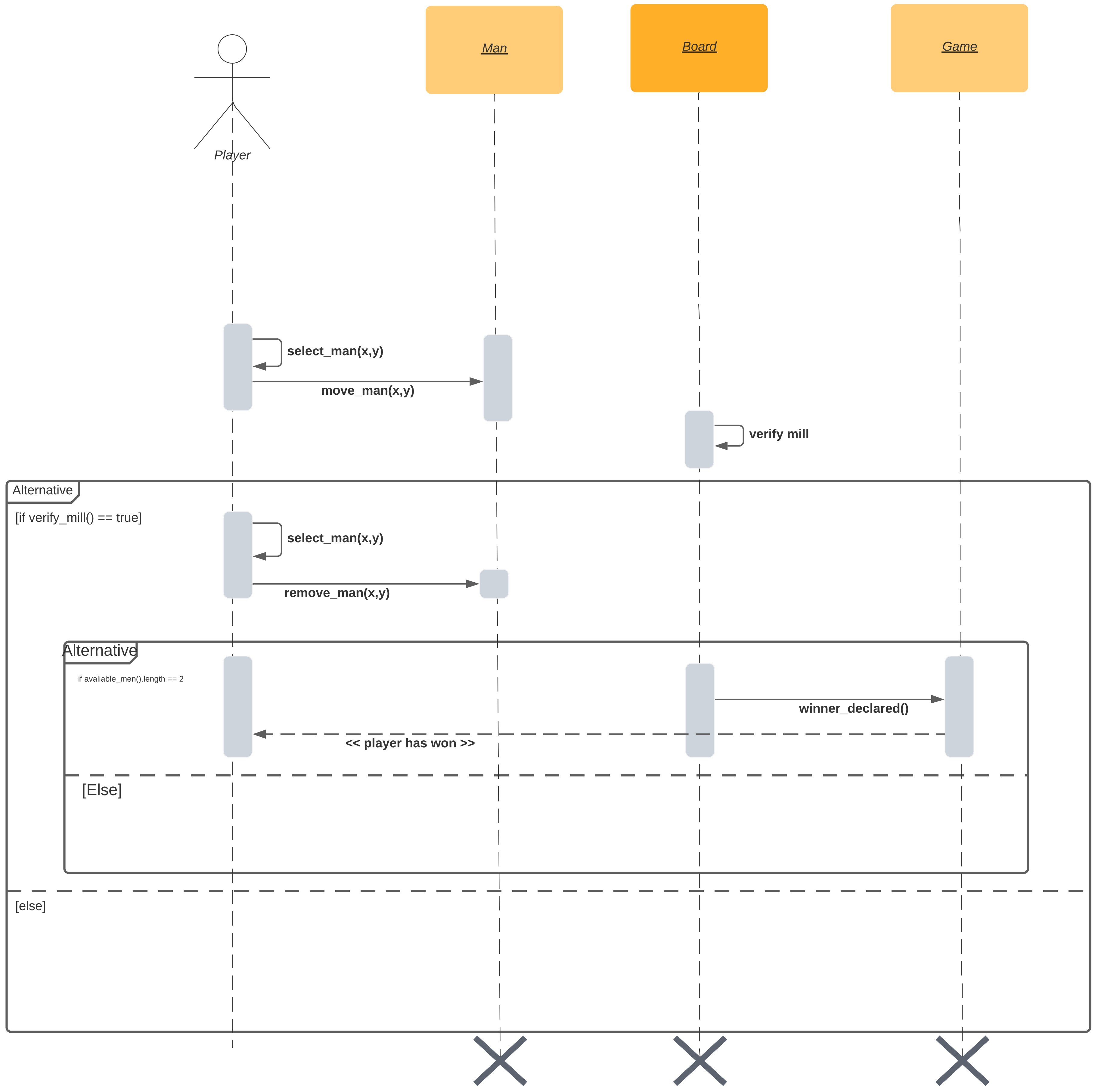


Player
Board
Man
available_intersections()
all available intersections
select_intersection()
is_intersection_occupied()
Alternative
[if is_intersection_occupied() == false]
false <<intersection not occupied>>
select_man()
move_man()
[Else]
true <<intersection occupied>>
<<choose another intersection >>
verify_mill()
Alternative
[if verify_mill() == true]
Loop
[while remove_man() == false]
select_man()
remove_man()
Alternative
[if remove_man() == false]
<<choose another opponent piece >>
[Else]
[Else]

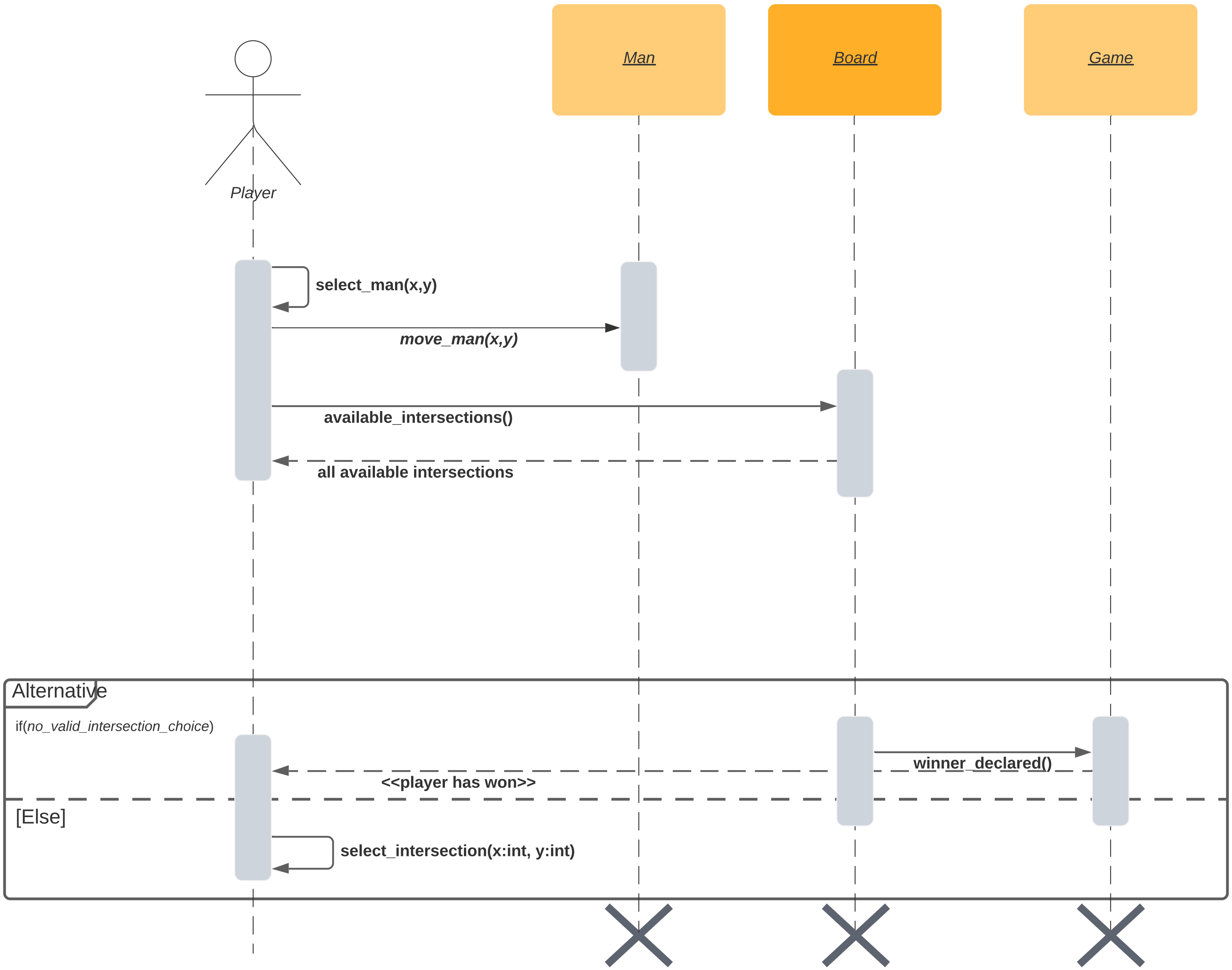
Use Case 3: Move



Use case 4: End game by removing pieces



Use Case 5: End game with no valid moves


Player
Man
Board
Game
select_man(x,y)
move_man(x,y)
available_intersections()
all available intersections
Alternative
if(no_valid_intersection_choice)
winner_declared()
<<player has won>>
[Else]
select_intersection(x:int, y:int)