Software Design Document

For

<u>Smart Chess Game</u>

Khanh Tran

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to show how the interactive chess game will be implemented as described in the Requirements document. The interactive chess game will allow two people to play a game of chess with two different machines anywhere with an internet connection.

1.2. Scope

This document will describe the application's architecture and how different components of the application interact with each other to achieve the requirements. However, this document will not go into how the testing process is performed.

1.3. Definitions, Acronyms, Abbreviations

1.3.1. Chess

- **Player:** A user who plays the game.
- **Board:** A term that describes an 8x8 matrix with brown and white colors on smaller squares.
- **Piece:** A term that describes an item on the chessboard, such as a king, queen, or pawn.
- **Check:** A term for a position when a king piece is put under the threat of being taken in the next turn.
- **Checkmate:** It is a term for a position when a king is put into check and there are no possible moves to get out. The game is declared over when this happens.
- **AI:** Artificial Intelligence.

1.3.2. Software

- **Host/Server:** A process that handles requests and returns responses to the clients.
- **Client/Guest:** A end-users' web browser run and render the game
- **IDE:** Integrated Development Environment, where developers edit and format the code.
- **UI:** User Interface.
- **OOP:** Object-Oriented Programming

2. Design Overview

2.1. Description of Problem

The interactive chess game application creates an environment for 2 users at remote locations to play chess together. First, the application needs to implement the rules of chess. Second, the application should be able to render the chessboard with chess pieces to the 2 players' screen. Last, the application should be able to connect to 2 players together and synchronize the move between the players with low latency.

2.2. Technologies Used

The interactive chess game can be used by any computer that has access to the internet and can render the HTML5 in the browser.

The interactive chess game web application uses Node.JS and Socket.IO as the back-end technology and HTML/CSS/JQuery as front-end technologies. Sock.IO enables real-time connection between 2 players in the game.

2.3. System Architecture

2.3.1. Overview

The application separates the system into different components so that each component can be developed independently without worrying about other parts. The request handlers from server and client agree on how to interact such as the kinds of request and the response format. Each component in the system specializes in a specific task which makes the system easier to add new functionalities in the future.

2.3.2. Server

2.3.2.1. Server Request Handler

The Server Request Handler is responsible for routing the request from clients to the corresponding subsystem in the server that is created to handle that request. After the request is handled by the proper subsystem, the Server Request Handler sends the response back to the client.

2.3.2.2. Room Manager

The Room Manager is responsible for managing the game rooms in the application. It can handle requests to create a new room, join an existing room, or get the game state of the room.

2.3.2.3. Game Manager

The Game Manager is the bridge between the request and the Chess API. It can interpret the request from the client and deliver the request to the Chess API.

2.3.2.4. Chess API

The Chess API offers various methods in order to verify the chess rules. It can check whether the move is valid or not, make a move if it is valid for the current chessboard, and get all available moves for a given piece in the chessboard.

2.3.2.5. Minimax

The Minimax object will represent the Easy Level of the AI game. It will make a move against the users based on an evaluation function, which consists of a piece-square values table.

2.3.2.6. Minimax with Alpha Beta Pruning

The Minimax with Alpha Beta Pruning object will represent the Medium Level of the AI game. It will make a move against the users based on an evaluation function, which consists of a piece-square values table. Furthermore, this object will prune branches that does not lead to the winning cases.

2.3.3. Client

2.3.3.1. Client Request Handler

The Client Request Handler is responsible for making the requests from the client to the server and delivers the response from the server to Renderer.

2.3.3.2. Screen Renderer

The Screen Renderer is responsible for rendering new changes to the screen. It can properly render the chessboard, other information related to the game, and in-game messages or errors from servers.

2.3.3.3. Event Handler

The Event Handler is responsible for handling the events that the users make from the screen (clicking mouse, pressing enter, etc.) The Event Handler interprets these events from the users and makes a corresponding request to the client request handler.

2.3.4. Deployment

There are 2 environments for this application: Development and Production. The application’s server will be deployed to Heroku as the Production instance. The Development instance can be hosted locally for testing and developing purposes.

Figure 1 shows the system architecture diagram for the application. It illustrates the relationship between different components in the system which enable the game to work.

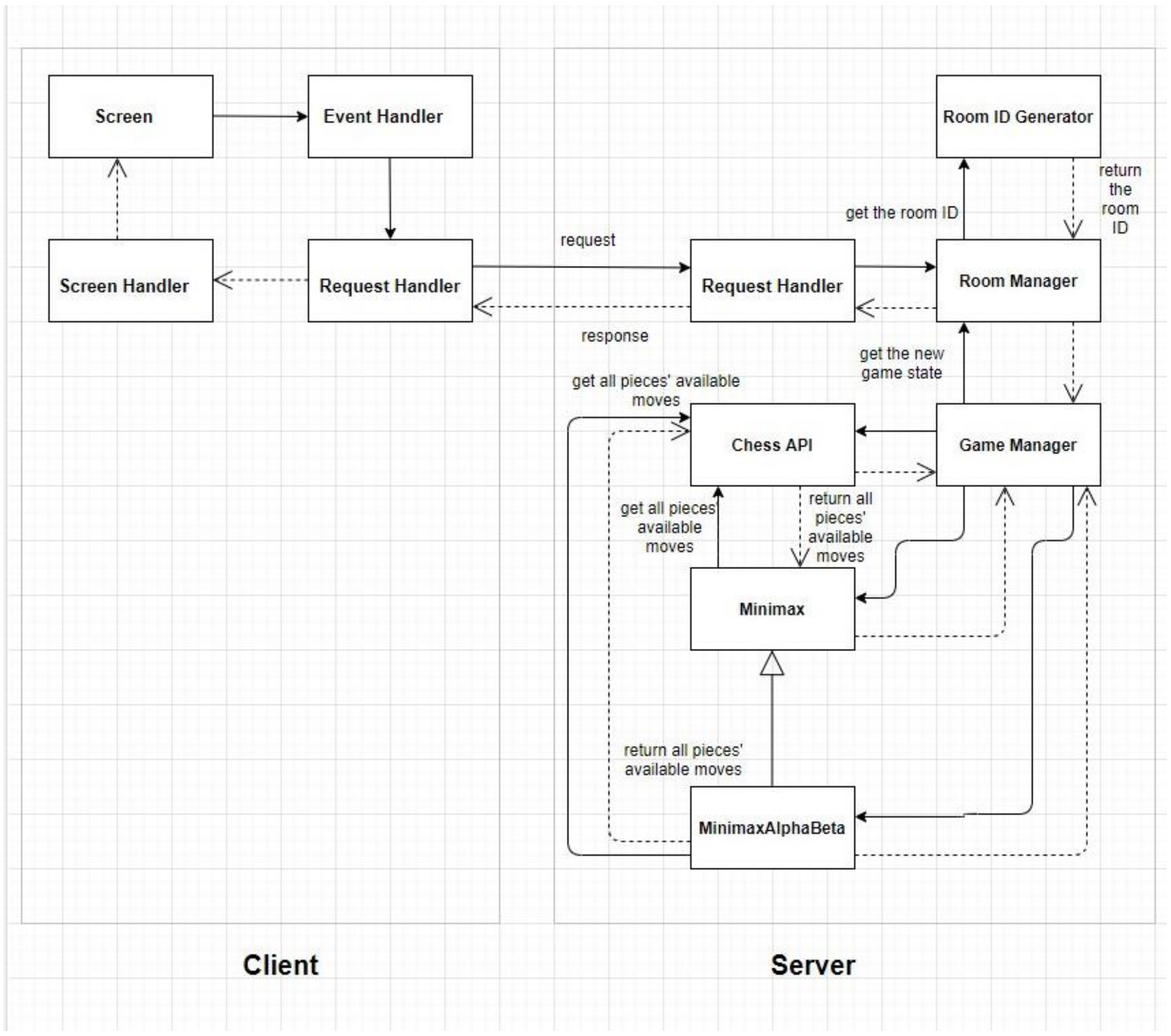

Figure 1: System Architecture for the Smart Chess Game

3. System Operations

Figure 2 shows the system operations in chronological order.

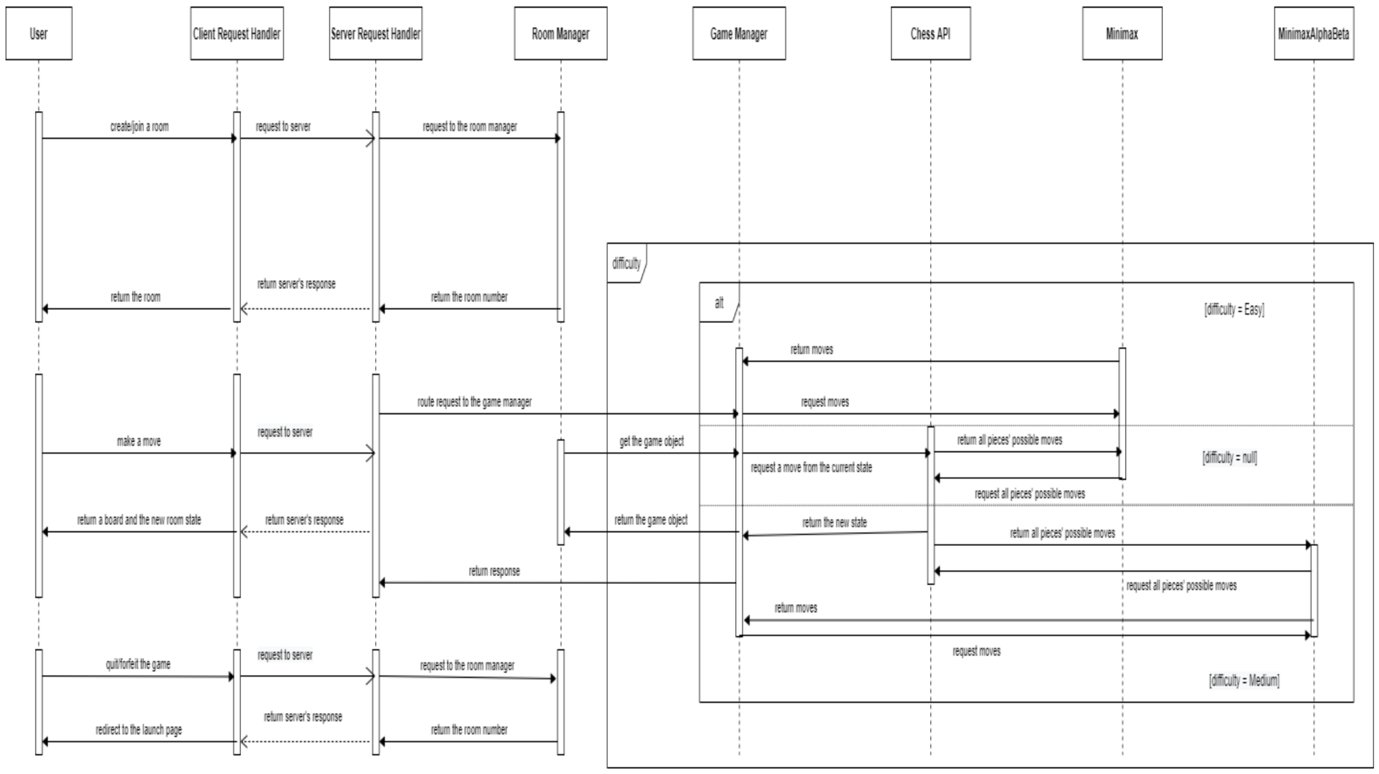

Figure 2: Sequence diagram for Smart Chess Game

3.1. Create/Join a game

The user starts by creating or joining a game room. The Client Request Handler then makes the request from the client to the server. The Server Request Handler then routes the request to the Room Manager. The Room Manager then performs the operation (create or join room) and return a corresponding response. The Request Handler sends the response back to the client. From the client-side, the Client Request Handler receives the response and routes the response to the Screen Renderer to render the room on the user's screen.

3.2. Make a move

The user starts by making a move (move a piece to a new position). The Client Request Handler then makes the request to the server. The Server Request Handler routes the request to the Game Manager, and the Game Manager sends the move to the Chess API. The Chess API sends the response to the Game Manager, Server Request Handler, and then Client Request Handler. Then, it gots to the Screen Renderer to render new changes in the game room on the screen.

3.3. Forfeit the game

The user starts by forfeiting the game. The Client Request Handler makes a request to the server which sends to the Room Manager through the Server Request Handler. The Room Manager updates the room and sends it back to the client. The user screen, in the end, is redirected to the launch page.

4. System Components Design

4.1. Server

4.1.1. Server Request Handler

- **Create a new room**

  This ends the request to the room manager and sends the response back to the client.

- **Join an existing room**

  Same as creating a new room.

- **Make a move**

  This operation sends the request to the game manager and sends the response back to the client.

- **Quit the room**

  Same as creating a new room.

4.1.2. Room Manager

- **Get a new room**

  This gets a new room ID and assigns the room to a user

- **<u>Get new room ID</u>**
  This randomly generates a new room ID which uses all upper and lower case letters and numbers). The room ID contains 5-7 characters.

- **<u>Join an existing room</u>**
  This checks if the room is full (already had 2 players). If not, this assigns the player to the room and returns. If the room is full, then return an error message.

- **<u>Get game state</u>**
  This returns the game state (the chess object) which contains all information related to the game.

4.1.3. Game Manager

- **<u>Make a move</u>**
  Parse and make the move request to the chess API.

4.1.4. Chess API

- **<u>Validate the move</u>**
  This validates if a move is valid with respect to the chess rules.
- **<u>Make a move</u>**
  This makes the move on the given board.
- **<u>Get available moves</u>**
  This gets all available moves of a given piece in the board.

4.1.5. Minimax

- **<u>Evaluate Moves</u>**
  This will calculate the score of each piece’s moves. The highest score will be used to determine the best move to go.
- **<u>Move</u>**
  The Minimax algorithm will move the piece to the specific position based on the evaluation.

4.1.6. MinimaxAlphaBeta

- **<u>Evaluate Moves</u>**

This will calculate the score of each piece’s moves. The highest score will be used to determine the best move to go.

- **<u>Move</u>**

  The Minimax Alpha Beta algorithm will move the piece to the specific position based on the evaluation. In this algorithm, we will prune out the branches that prove the move to be worse than a previously examined one.

Figure 3 below shows the OOP class diagram and the relationship between different components in the server.

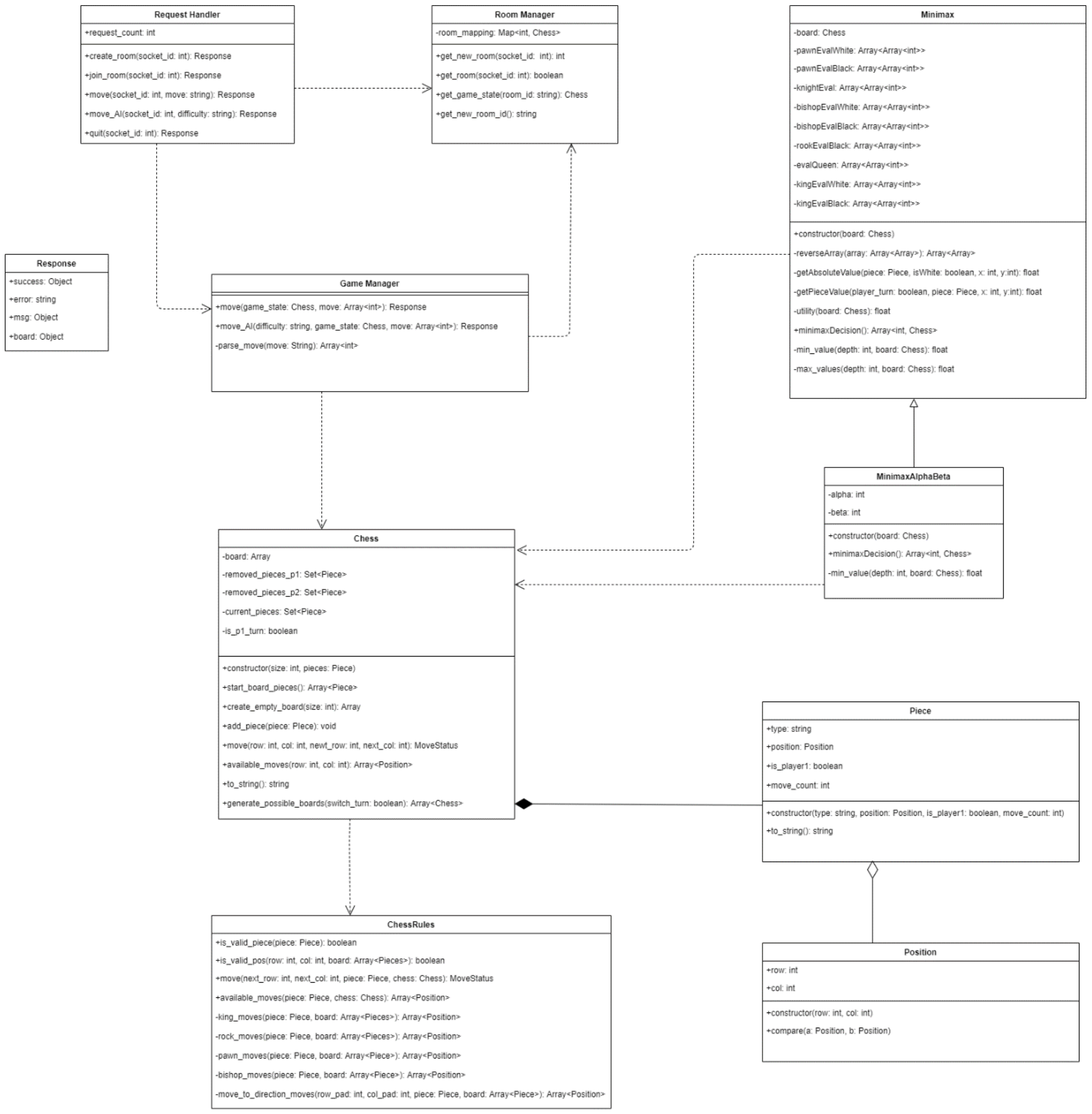
Request Handler
+request_count: int
+create_room(socket_id: int): Response
+join_room(socket_id: int): Response
+move(socket_id: int, move: string): Response
+move_AI(socket_id: int, difficulty: string): Response
+quit(socket_id: int): Response
Room Manager
-room_mapping: Map<int, Chess>
+get_new_room(socket_id: int): int
+get_room(socket_id: int): boolean
+get_game_state(room_id: string): Chess
+get_new_room_id(): string
Minimax
-board: Chess
-pawnEvalWhite: Array<Array<int>>
-pawnEvalBlack: Array<Array<int>>
-knightEval: Array<Array<int>>
-bishopEvalWhite: Array<Array<int>>
-bishopEvalBlack: Array<Array<int>>
-rookEvalBlack: Array<Array<int>>
-evalQueen: Array<Array<int>>
-kingEvalWhite: Array<Array<int>>
-kingEvalBlack: Array<Array<int>>
+constructor(board: Chess)
-reverseArray(array: Array<Array>): Array<Array>
-getAbsoluteValue(piece: Piece, isWhite: boolean, x: int, y:int): float
-getPieceValue(player_turn: boolean, piece: Piece, x: int, y:int): float
-utility(board: Chess): float
+minimaxDecision(): Array<int, Chess>
-min_value(depth: int, board: Chess): float
-max_values(depth: int, board: Chess): float
Response
+success: Object
+error: string
+msg: Object
+board: Object
Game Manager
+move(game_state: Chess, move: Array<int>): Response
+move_AI(difficulty: string, game_state: Chess, move: Array<int>): Response
-parse_move(move: String): Array<int>
MinimaxAlphaBeta
-alpha: int
-beta: int
+constructor(board: Chess)
+minimaxDecision(): Array<int, Chess>
-min_value(depth: int, board: Chess): float
Chess
-board: Array
-removed_pieces_p1: Set<Piece>
-removed_pieces_p2: Set<Piece>
-current_pieces: Set<Piece>
-is_p1_turn: boolean
+constructor(size: int, pieces: Piece)
+start_board_pieces(): Array<Piece>
+create_empty_board(size: int): Array
+add_piece(piece: Piece): void
+move(row: int, col: int, newt_row: int, next_col: int): MoveStatus
+available_moves(row: int, col: int): Array<Position>
+to_string(): string
+generate_possible_boards(switch_turn: boolean): Array<Chess>
Piece
+type: string
+position: Position
+is_player1: boolean
+move_count: int
+constructor(type: string, position: Position, is_player1: boolean, move_count: int)
+to_string(): string
ChessRules
+is_valid_piece(piece: Piece): boolean
+is_valid_pos(row: int, col: int, board: Array<Pieces>): boolean
+move(next_row: int, next_col: int, piece: Piece, chess: Chess): MoveStatus
+available_moves(piece: Piece, chess: Chess): Array<Position>
-king_moves(piece: Piece, board: Array<Pieces>): Array<Position>
-rock_moves(piece: Piece, board: Array<Pieces>): Array<Position>
-pawn_moves(piece: Piece, board: Array<Piece>): Array<Position>
-bishop_moves(piece: Piece, board: Array<Piece>): Array<Position>
-move_to_direction_moves(row_pad: int, col_pad: int, piece: Piece, board: Array<Piece>): Array<Position>
Position
+row: int
+col: int
+constructor(row: int, col: int)
+compare(a: Position, b: Position)

Figure 3: The Server Class Diagram

4.2. Client

4.2.1. Client Request Handler

- **<u>Create a new game</u>**
  This makes a request to the server to create a new game.

- **<u>Join an existing game</u>**

This makes a request to the server to join an existing game.

- **<u>Make a move</u>**
  On the client-side, they select a square on the board. The client sends the x and y coordinates selected on the board to the server. This makes a request to the server to make a move.

- **<u>Quit game</u>**
  This makes a request to the server to quit the game.

- **<u>Room update handler</u>**
  This handles the response to update room (game state) from the server.

4.2.2. Screen Renderer

- **<u>Render board</u>**
  This renders the board on the screen.

- **<u>Render room</u>**
  This renders all information related to the room (player’s turn, score, etc.)

- **<u>Display message</u>**
  This renders all related messages from the server to the screen.

4.2.3. Event Handler

- **<u>Button clicked</u>**
  This handles the button click and sends the proper request to the client request handler.

- **<u>Piece clicked</u>**
  This handles the piece click and sends the proper request to the client request handler. The board displays all possible moves.

Figure 4 shows the OOP class diagram and the relationship between different components in the client.

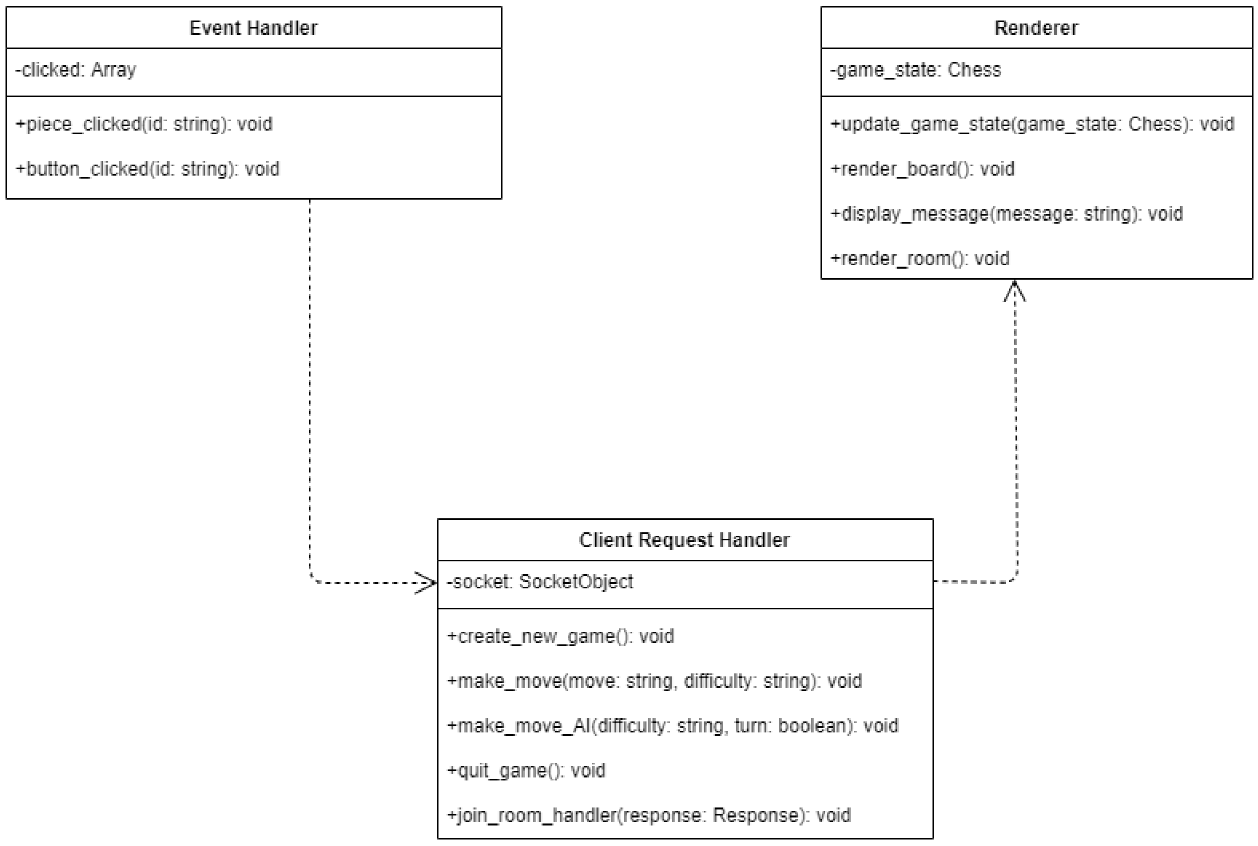
Event Handler
-clicked: Array
+piece_clicked(id: string): void
+button_clicked(id: string): void
Renderer
-game_state: Chess
+update_game_state(game_state: Chess): void
+render_board(): void
+display_message(message: string): void
+render_room(): void
Client Request Handler
-socket: SocketObject
+create_new_game(): void
+make_move(move: string, difficulty: string): void
+make_move_AI(difficulty: string, turn: boolean): void
+quit_game(): void
+join_room_handler(response: Response): void

Figure 4: Client Class Diagram

5. System Scalability

Although the system does not involve storing user data in a database, the server handles users' requests and stores user's socket connections and the game state for each room. This will become the bottleneck of the system when the number of users playing the game at the same time increases. At some point when the room increases, the server runs out of resources (memory and CPU) to handle all requests coming from the clients.

We can scale the system vertically by adding more memory and computational power. However, there will be a limit at which the computer cannot handle more requests, and the performance starts to degrade.

Therefore, we should scale the system horizontally by adding more servers and machines. Multiple instances of servers will be able to serve more requests from users. We would need to add a load balancer to balance the load between multiple servers. For multiple servers, there are many new problems such as how to recover and handle failure when a server goes down, how to make sure 2 players get the same state, etc.

There are some options like Kubernetes and Mesos which are systems for automating deployment, scaling, and management of applications. These platforms will help scale out the application by automatically creating or removing new nodes on demand for availability and redundancy.

6. User Interface Design

6.1. Overview

In this section, more details about the implementation of components will be mentioned. The descriptions are illustrated as UML diagrams and mock-ups.

6.2. Board Design

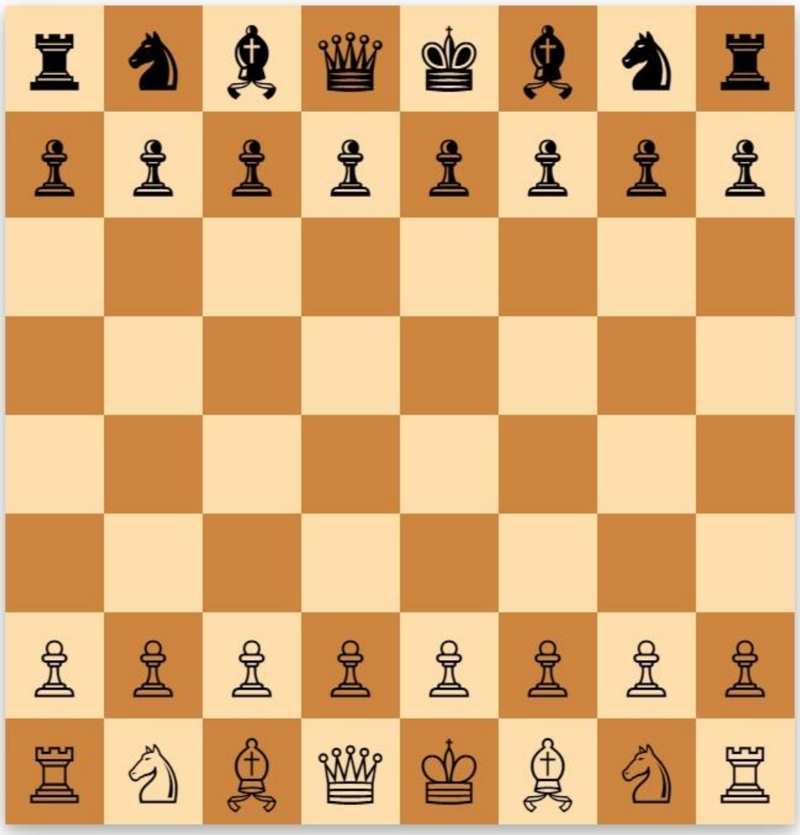
Figure 5: An 8x8 brow and white chessboard with all pieces

Important Functions:

- Create a 8x8 empty matrix
- Instantiate all chess pieces
- Make move on the piece, which is selected by players
- Check if there is any piece in the space
- Display available moves of the piece

6.3. Pieces

Figure 6: Black pieces

Figure 7: White pieces

Chess pieces are illustrated as black and white pieces in Figure 6 and 7. Below are critical methods that will be used in Piece class:

- Piece position
- Name of players
- Type of players (black or white)

7. Launch Page Interface

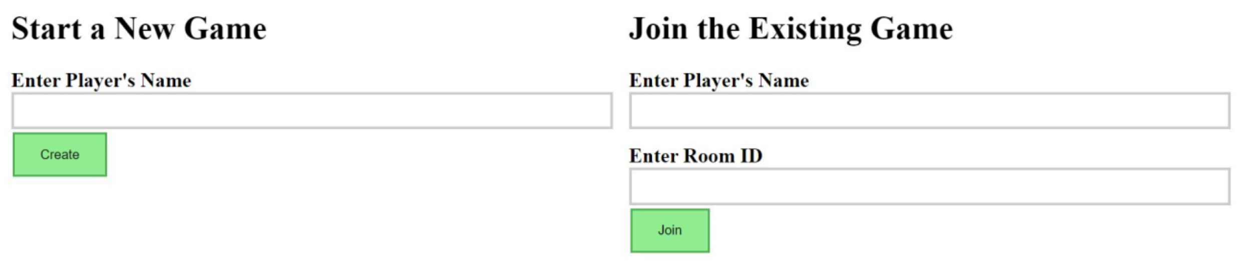

Figure 8: Launch Page UI

The first user will enter his or her name, and click on the "Create" button to initialize a new chess game, with a specific ID for this game. After entering his or her name, the second player will contact the first player to get the ID for the game. If the ID is correct, the second player will join the game with the first player. Once the chess game has two people, another person cannot join the game.