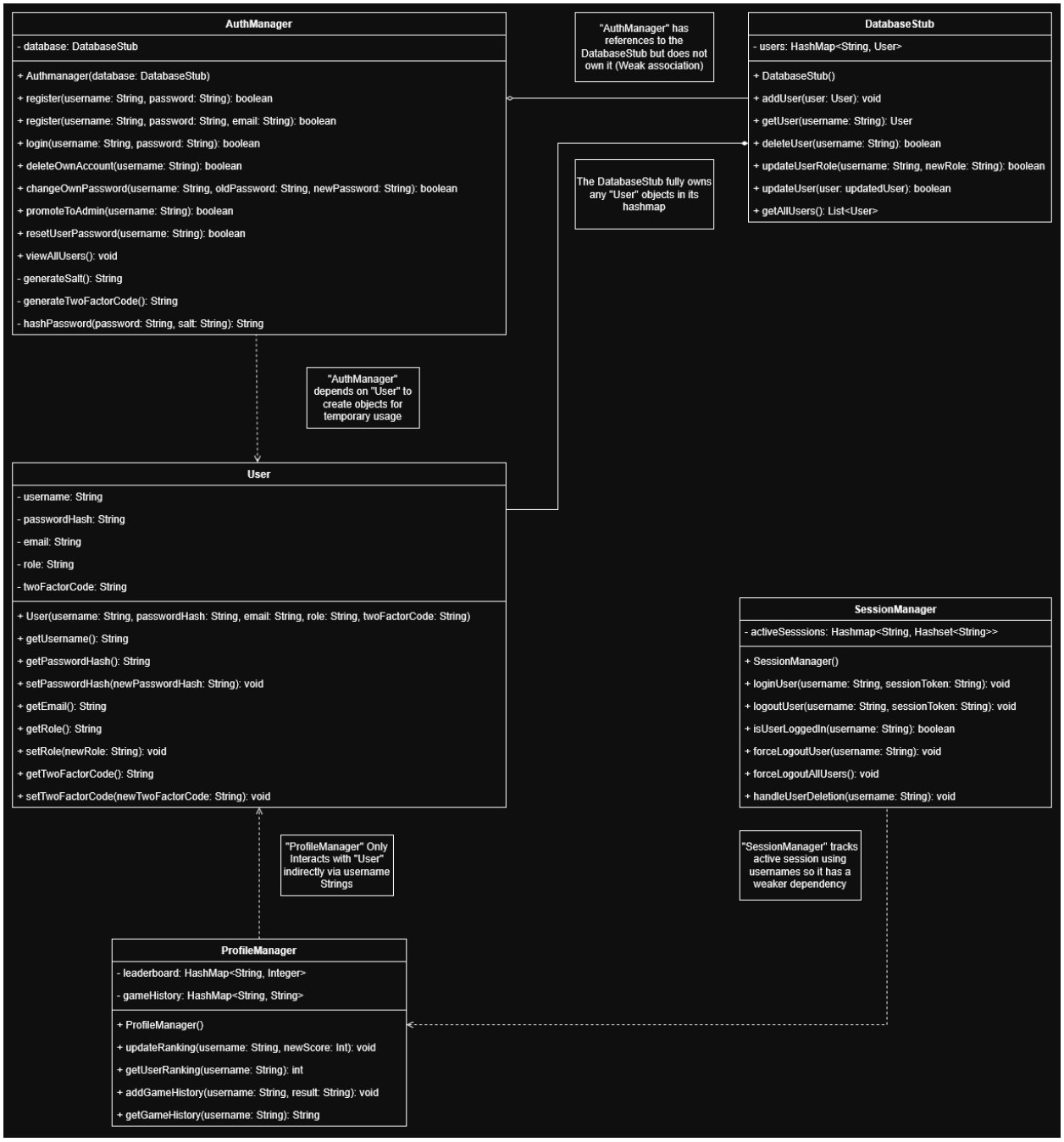
Focuses on reviewing diagrams, structure, and core system design.
Planning Analysis Sub-Team.

Files under review:

- ★ Class Structure Diagrams
  - Authentication design
  - Gamelogic design
  - Leaderboard design
  - Networking design
  - Matchmaking designs
  - GUI designs
- ★ Use Case Diagrams
  - Authentication design
  - Gamelogic designs
  - Matchmaking design
  - Networking design
  - GUI designs
- ★ Use Case Descriptions
  - Platform design
  - Authentication design
  - Profile design
  - Leaderboard design
  - Networking design
  - Gamelogic Designs
- ★ UI mockups
  - Platform design
  - Authentication design
  - Profile design
  - Leaderboard design
  - Networking design
  - Gamelogic Designs

Class Structure Diagrams

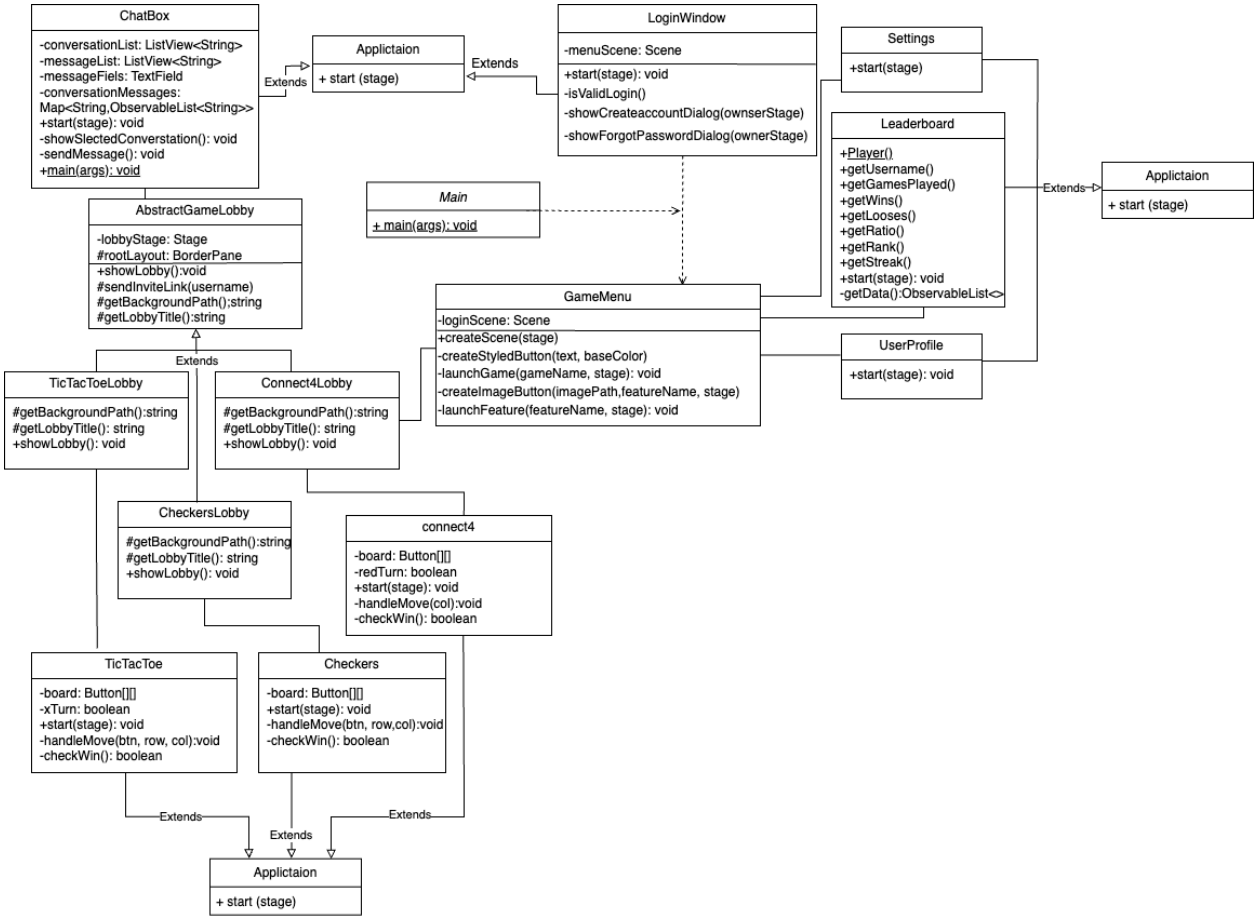
Authentication design:



Areas for Improvement in Authentication design:

1. **`AuthManager` Password Handling:**
    - The `hashPassword(password, salt)` method should return a `byte[]` instead of `String`, as password hashes should not be stored as plain strings.
    - Consider adding a `verifyPassword(inputPassword, storedHash, salt)` method for authentication instead of directly comparing hashes.
2. **Lack of Associations in `ProfileManager`:**
    - `ProfileManager` relies only on usernames. If `User` objects need to be retrieved often, consider a reference to `DatabaseStub` instead of just storing usernames.
3. **Missing Constraints in `SessionManager`:**
    - Does `SessionManager` limit the number of concurrent sessions per user?
    - Adding a max session limit per user would improve security.
4. **`DatabaseStub` User Modification Methods:**
    - `updateUserRole` modifies a user's role but does not specify if it verifies existing roles or permissions.
    - `updateUser` should clarify whether it replaces an entire `User` object or updates specific fields.
5. **Two-Factor Authentication Storage (`User` Class):**
    - The `twoFactorCode` is stored as a string, which may be a security risk. Consider an expiration time or a mechanism to invalidate codes.

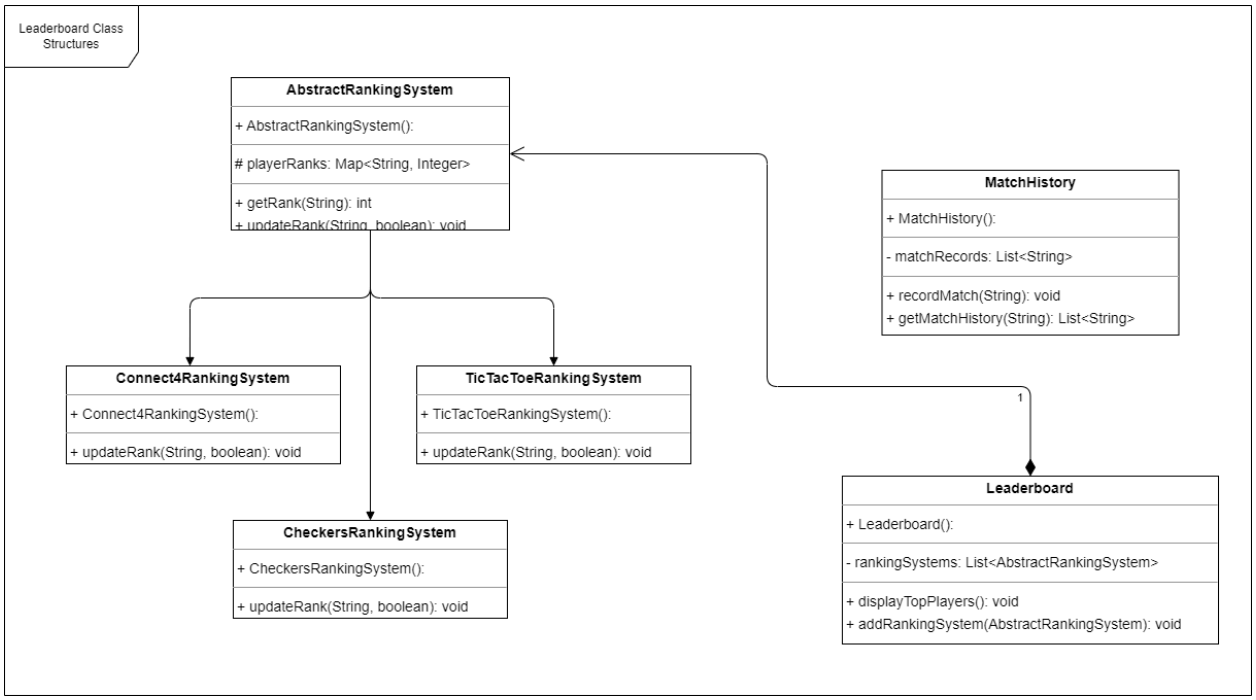
Gamelogic Designs



Areas for Improvement in Game-logic Design:

1. **Lack of a Clear Inheritance Structure for Game Logic Classes**
    - The `TicTacToeLogic`, `Connect4Logic`, and `CheckersLogic` classes do not inherit from a common abstract class or interface. A `GameLogic` interface or an abstract class could define shared methods like `startGame()`, `endGame()`, and `win()` to enforce consistency.
2. **GUI and Logic Should Be More Decoupled**
    - The diagram suggests a direct link between the GUI classes (`TicTacToeGUI`, `Connect4GUI`, `CheckersGUI`) and the logic classes. It would be better to have a controller or mediator class to handle communication between the GUI and game logic.
3. **AbstractBoard Could Be Better Utilized**
    - `TicTacToeBoard`, `Connect4Board`, and `CheckersBoard` do not inherit from `AbstractBoard`. Since `AbstractBoard` already contains a `board` attribute, these classes could extend it to reduce code duplication.
4. **CheckerPiece Should Inherit from AbstractPiece**
    - The `CheckerPiece` class has similar attributes (`xPos`, `yPos`, `colour`) as `AbstractPiece`. It should extend `AbstractPiece` instead of defining these attributes again.
5. **Main Class Should Not Directly Control GUI**
    - The `Main` class appears to link directly to GUI components. A better design would use a `GameManager` class to coordinate game initialization and logic, while the GUI updates based on observer patterns or event-driven design.

Leaderboard design



Strengths and Weaknesses of the Leaderboard Class Diagram

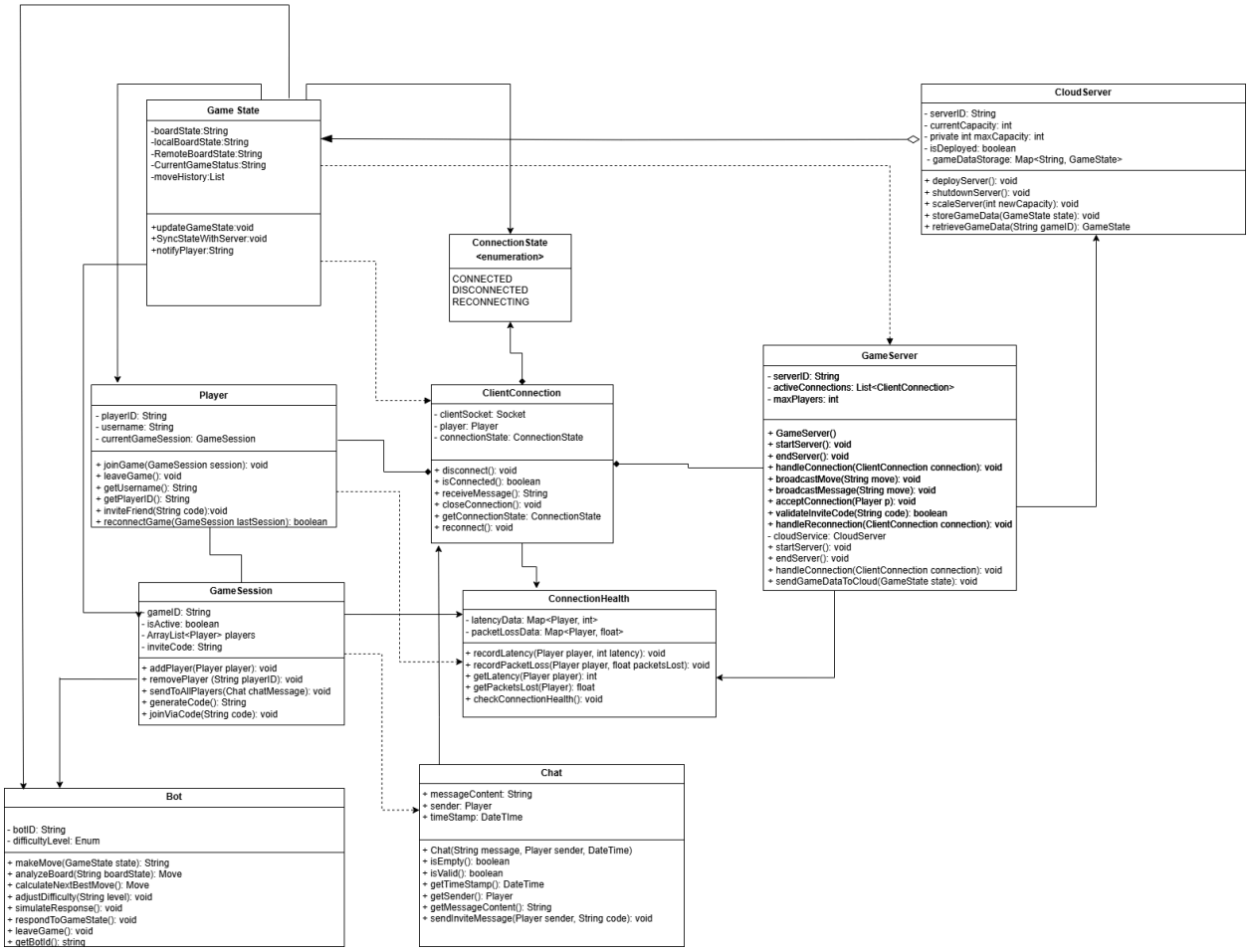
Strengths:

1. **Modular and Extensible Ranking System:**
    - The `AbstractRankingSystem` class provides a solid base for different game ranking systems (`TicTacToeRankingSystem`, `Connect4RankingSystem`, and `CheckersRankingSystem`). This makes it easy to extend ranking for additional games in the future.
2. **Separation of Concerns:**
    - The diagram follows a well-structured approach where ranking, leaderboard management, and match history are handled by separate classes (`Leaderboard`, `MatchHistory`, and ranking system classes). This ensures that each class has a single, focused responsibility.
3. **Encapsulation of Player Data:**
    - Player ranks are stored within the `AbstractRankingSystem` using a `Map<String, Integer>`, ensuring efficient retrieval and updates while maintaining encapsulation. The use of `updateRank(String, boolean)` helps in rank modifications without direct data manipulation.

Weaknesses:

1. **Lack of Direct Player Association:**
    - The ranking system does not explicitly link to a `Player` class. If a player object existed, it would allow for more detailed attributes such as usernames, stats, and game preferences, improving functionality.
2. **Leaderboard Lacks Sorting Mechanism:**
    - While `displayTopPlayers()` exists, there is no indication of how the leaderboard sorts and ranks players based on their game performance. A sorting mechanism based on ranking scores should be clearly defined.
3. **MatchHistory Class Could Be More Integrated:**
    - The `MatchHistory` class exists separately, but there is no direct relationship between it and the ranking system. Integrating match records with rank updates would ensure a more comprehensive tracking system for performance evaluation.

Networking design



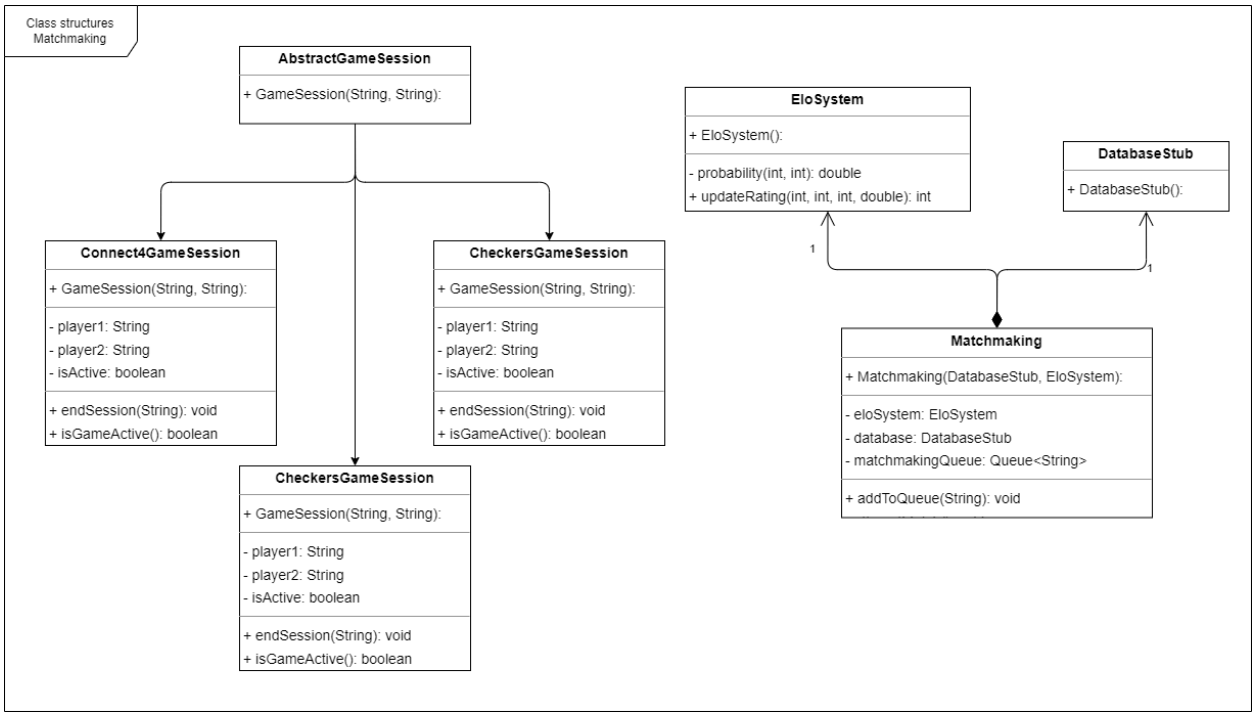
Strengths and Weaknesses of the Networking Class Diagram

1. **Well-Defined Player and Connection Handling**
    - The `Player` class includes methods for joining, leaving, and reconnecting to a `GameSession`, ensuring **robust session management**.
    - `ClientConnection` and `ConnectionState` provide a **clear structure for managing player connectivity**, allowing for states like `CONNECTED`, `DISCONNECTED`, and `RECONNECTING`.
2. **Good Server-Cloud Interaction**
    - The `GameServer` can send game data to the `CloudServer`, which stores game states (`retrieveGameData`, `storeGameData`). This provides **scalability** and **persistence** for online gaming sessions.
3. **Comprehensive Network Health Monitoring**
    - `ConnectionHealth` tracks **latency and packet loss**, providing methods to evaluate player connection quality (`recordLatency`, `getPacketLoss`, `checkConnectionHealth`).
    - This helps maintain a **smooth multiplayer experience** by adapting to connection issues.

Weaknesses

1. **No Clear Game Logic Separation**
    - The `GameSession` and `GameState` classes handle game-related data, but **game rules and mechanics (e.g., valid moves, win conditions)** are not explicitly represented.
    - A **separate `GameLogic` class** could help manage game rules efficiently.
2. **Limited Bot Functionality**
    - The `Bot` class provides functions like `makeMove` and `respondToGameState`, but **it lacks AI decision-making depth** (e.g., strategy, difficulty levels).
    - Adding **adaptive decision-making algorithms** could enhance bot performance.
3. **Potential Redundancy in Server Management**
    - Both `GameServer` and `CloudServer` handle storage (`storeGameData` and `sendGameDataToCloud`). However, **it's unclear why both are needed** instead of a **single backend service**.
    - **Merging or clarifying responsibilities** could make the system more efficient.

Matchmaking Designs



Strengths and Areas for Improvement in the Matchmaking Design

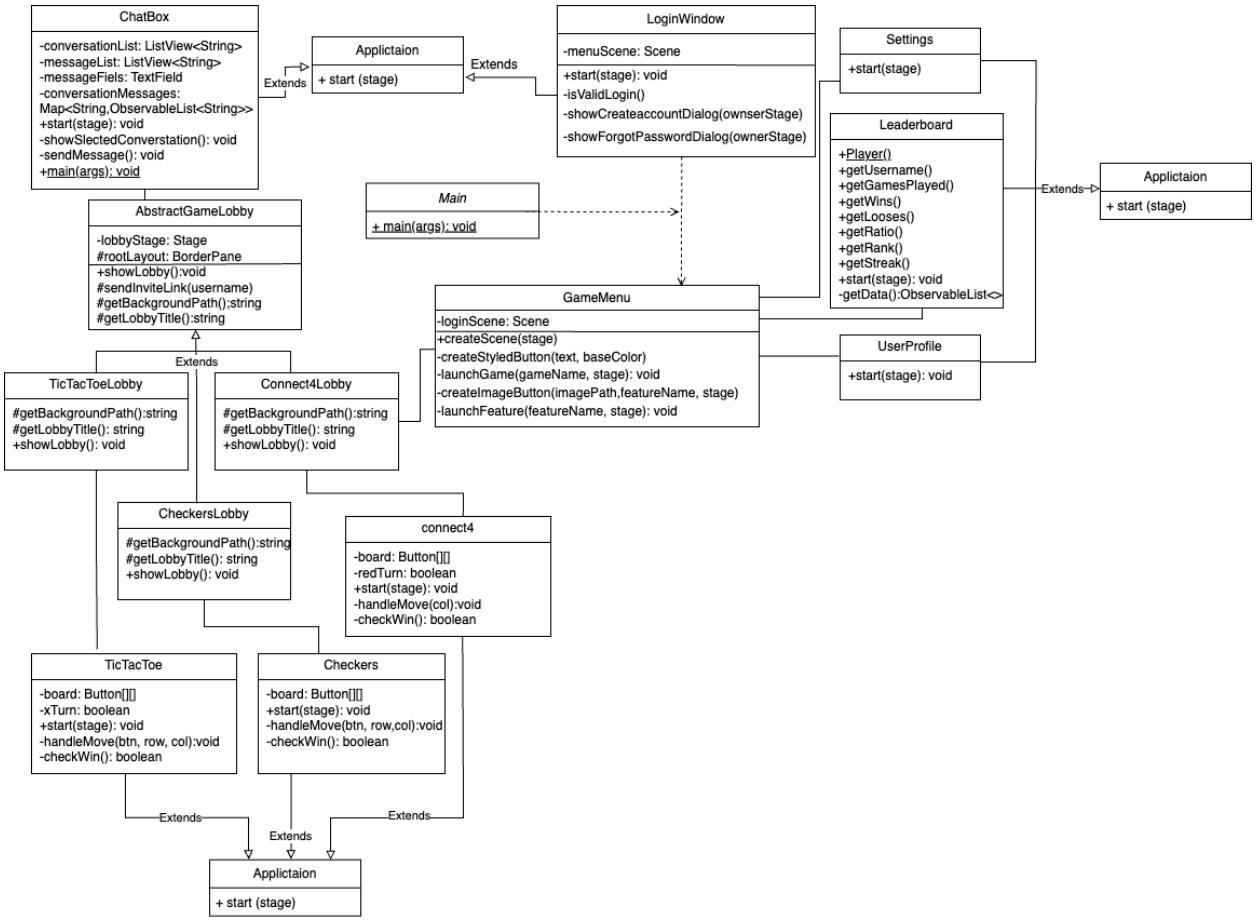
Strengths

1. **Clear Hierarchy & Inheritance Structure**
    - The diagram effectively uses **inheritance** for game sessions, ensuring reusability. `AbstractGameSession` serves as a parent class, reducing code duplication in `Connect4GameSession` and `CheckersGameSession`.
2. **Separation of Concerns**
    - Different functionalities are well-separated. `EloSystem` handles ranking logic, `DatabaseStub` represents database interactions, and `Matchmaking` manages player queues. This makes the system **modular and easy to maintain**.
3. **Proper Use of Associations**
    - The **one-to-one association** between `Matchmaking`, `EloSystem`, and `DatabaseStub` is well-defined, clarifying their roles in player matching.

Weaknesses

1. **Duplicate CheckersGameSession Class**
    - There are **two instances of `CheckersGameSession`** in the diagram. This is likely an error and should be fixed to avoid confusion.
2. **Limited Matchmaking Functionality**
    - `Matchmaking` only has `addToQueue(String)`, but **no method for actually matching players** or starting a game session. A method like `findMatch()` would improve completeness.
3. **No Clear Relationship Between Matchmaking and Game Sessions**
    - There is **no connection between `Matchmaking` and `AbstractGameSession` or its subclasses**. How does matchmaking initiate a game session? A direct link or a factory pattern could help clarify this interaction.

GUI designs


ChatBox
-conversationList: ListView<String>
-messageList: ListView<String>
-messageFiels: TextField
-conversationMessages: Map<String,ObservableList<String>>
+start(stage): void
-showSlectedConverstation(): void
-sendMessage(): void
+main(args): void
Extends
Applictaion
+ start (stage)
Extends
LoginWindow
-menuScene: Scene
+start(stage): void
-isValidLogin()
-showCreateaccountDialog(ownserStage)
-showForgotPasswordDialog(ownerStage)
Settings
+start(stage)
Leaderboard
+Player()
+getUsername()
+getGamesPlayed()
+getWins()
+getLooses()
+getRatio()
+getRank()
+getStreak()
+start(stage): void
-getData():ObservableList<>
Extends
Applictaion
+ start (stage)
Main
+ main(args): void
AbstractGameLobby
-lobbyStage: Stage
#rootLayout: BorderPane
+showLobby():void
#sendInviteLink(username)
#getBackgroundPath();string
#getLobbyTitle():string
GameMenu
-loginScene: Scene
+createScene(stage)
-createStyledButton(text, baseColor)
-launchGame(gameName, stage): void
-createImageButton(imagePath,featureName, stage)
-launchFeature(featureName, stage): void
UserProfile
+start(stage): void
Extends
TicTacToeLobby
#getBackgroundPath():string
#getLobbyTitle(): string
+showLobby(): void
Connect4Lobby
#getBackgroundPath():string
#getLobbyTitle(): string
+showLobby(): void
CheckersLobby
#getBackgroundPath():string
#getLobbyTitle(): string
+showLobby(): void
connect4
-board: Button[][]
-redTurn: boolean
+start(stage): void
-handleMove(col):void
-checkWin(): boolean
TicTacToe
-board: Button[][]
-xTurn: boolean
+start(stage): void
-handleMove(btn, row, col):void
-checkWin(): boolean
Checkers
-board: Button[][]
+start(stage): void
-handleMove(btn, row,col):void
-checkWin(): boolean
Extends
Extends
Extends
Applictaion
+ start (stage)

Strengths and Areas for Improvement in the Gui-Design

Strengths:

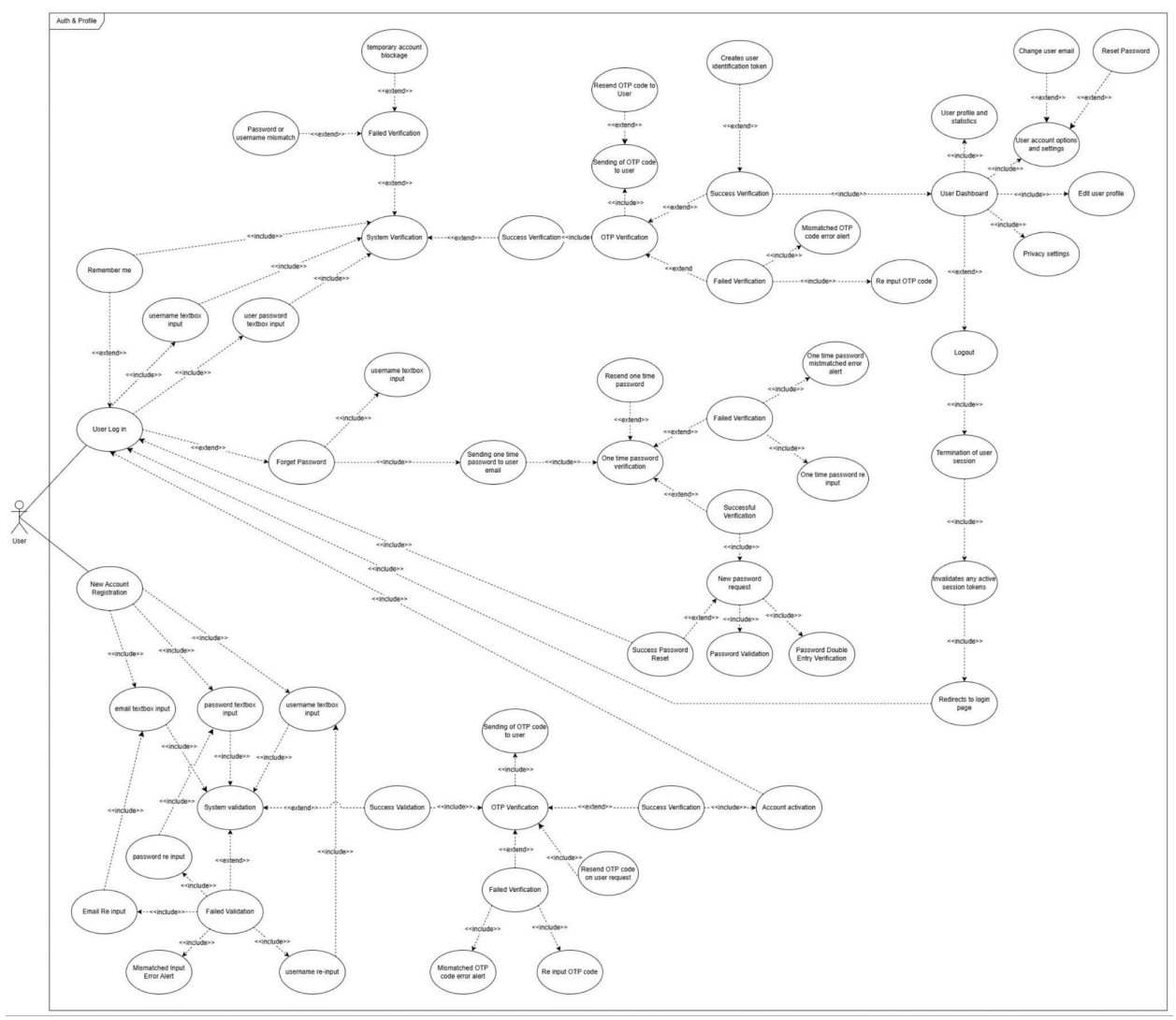
1. **Well-Structured Game Lobby Hierarchy:**
    - The `AbstractGameLobby` class is effectively used as a base class, with `TicTacToeLobby`, `Connect4Lobby`, and `CheckersLobby` extending it. This reduces redundancy and promotes code reuse.
2. **Separation of Concerns:**
    - The diagram clearly separates different functionalities, such as game logic (`TicTacToe`, `Checkers`, `connect4`), UI components (`LoginWindow`, `GameMenu`), and additional features (`Leaderboard`, `Settings`, `UserProfile`).
3. **Leaderboard System for Tracking Player Performance:**
    - The `Leaderboard` class includes methods for retrieving player statistics like `getWins()`, `getRatio()`, and `getRank()`, which enhances the competitive aspect of the system.

Areas for Improvement:

1. **Lack of a Base Game Class for Core Logic:**
    - `TicTacToe`, `Checkers`, and `connect4` all define similar attributes (e.g., `board: Button[][]`, `checkWin()`). A common `AbstractGame` class should be introduced to handle shared logic.
2. **Redundant Methods Across Games:**
    - Methods like `handleMove()` and `checkWin()` appear in multiple game classes. These could be moved to a parent `AbstractGame` class to improve maintainability.
3. **ChatBox Class Could Be More Modular:**
    - The `ChatBox` class has several responsibilities, such as managing messages and conversations. It could be split into smaller classes like `ConversationManager` and `MessageHandler` to improve maintainability.

Use Case Diagrams

Authentication design:



The use case diagram presents a structured overview of user interactions with an authentication and profile management system. However, it has several weaknesses:

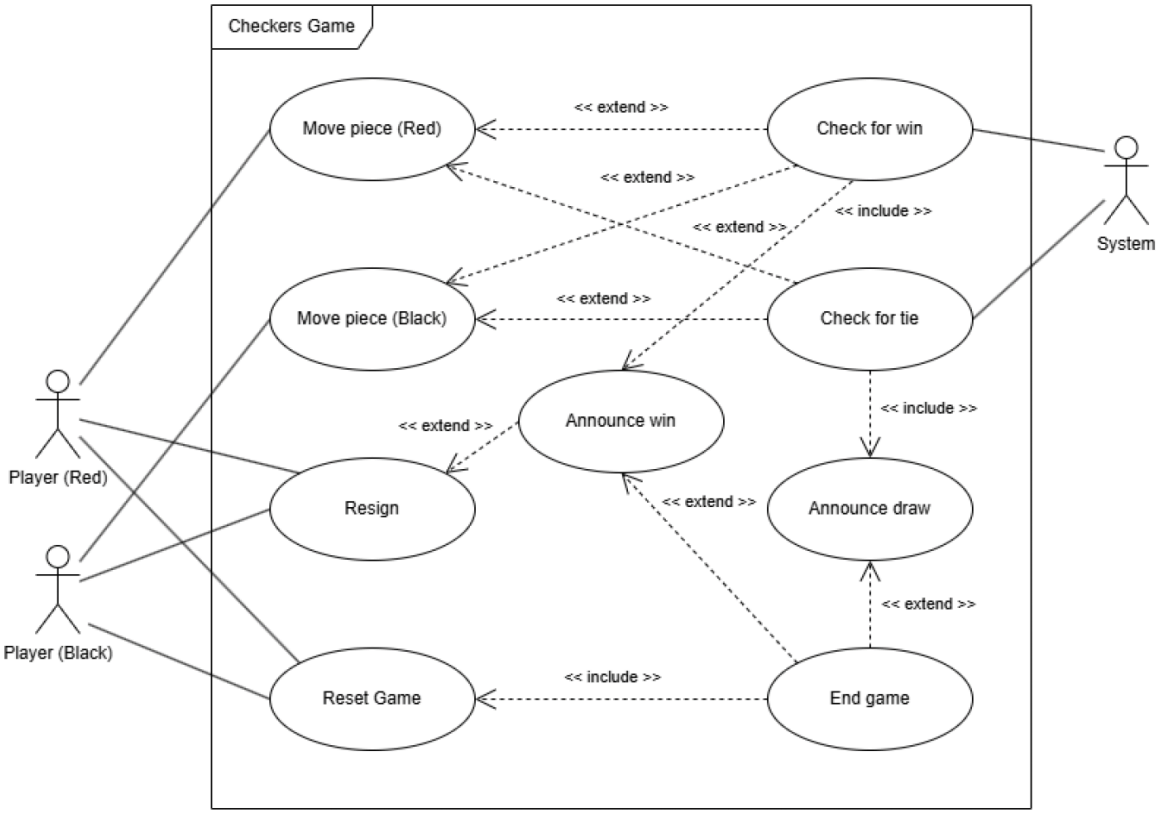
1. **Complexity and Clutter**:
    - The diagram is highly dense, making it difficult to follow the relationships between use cases.
    - Overlapping and closely spaced elements reduce readability.
2. **Lack of Modularization**:
    - The diagram attempts to depict too many use cases in a single view, rather than breaking it down into smaller diagrams focusing on login, registration, password recovery, and profile management separately.
3. **Redundant Use Cases**:
    - Some processes, like OTP verification and password reset, appear multiple times with slight variations, which could be generalized into a single reusable use case.
4. **Weak Use of Generalization and Includes/Extends**:
    - Some use cases, like "OTP Verification," are repetitive but not properly modularized using generalization.
    - The relationship between certain cases (e.g., "Forgot Password" and "Resend OTP") could be better clarified with `<<include>>` and `<<extend>>` relationships.
5. **Unclear Actor Responsibilities**:
    - The only actor present is "User," but system roles such as "Admin" (for account approval or monitoring) are absent.
    - The system itself should be represented in some cases where automatic verifications occur.
6. **Unnecessary Details for a High-Level Diagram**:
    - Some low-level details, such as "One-time password input" and "Fix input OTP code," could be better represented in an activity diagram rather than a use case diagram.

Recommendations for Improvement:

- **Break down the diagram into multiple focused diagrams** (Login, Registration, Password Management, etc.).
- **Use `<<include>>` and `<<extend>>` relationships effectively** to eliminate redundancy.
- **Simplify and remove unnecessary low-level details** that belong in process flow diagrams.
- **Introduce additional actors**, such as an admin or authentication system, to clarify responsibilities.
- **Improve spacing and organization** to enhance readability.

Gamelogic design

USE CASE DIAGRAM AND DESCRIPTIONS - CHECKERS



This is a **well-structured use case diagram** with clear relationships, but minor refinements could improve clarity and correctness.

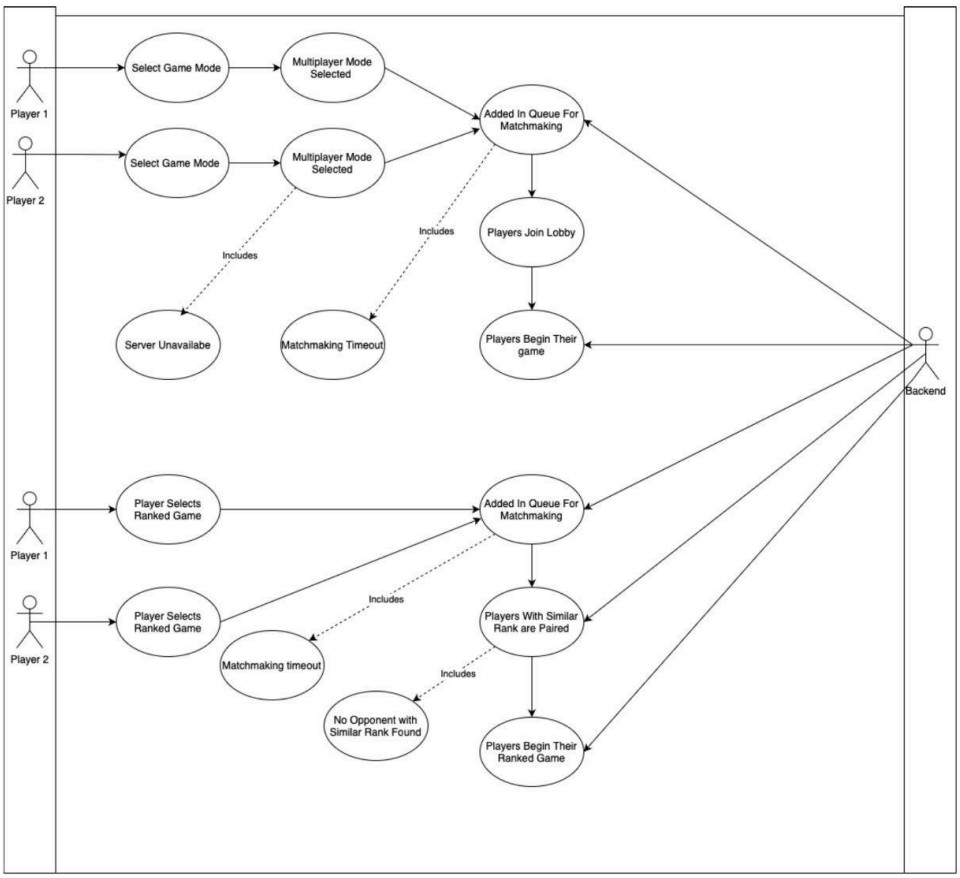
Suggestions for Improvement:

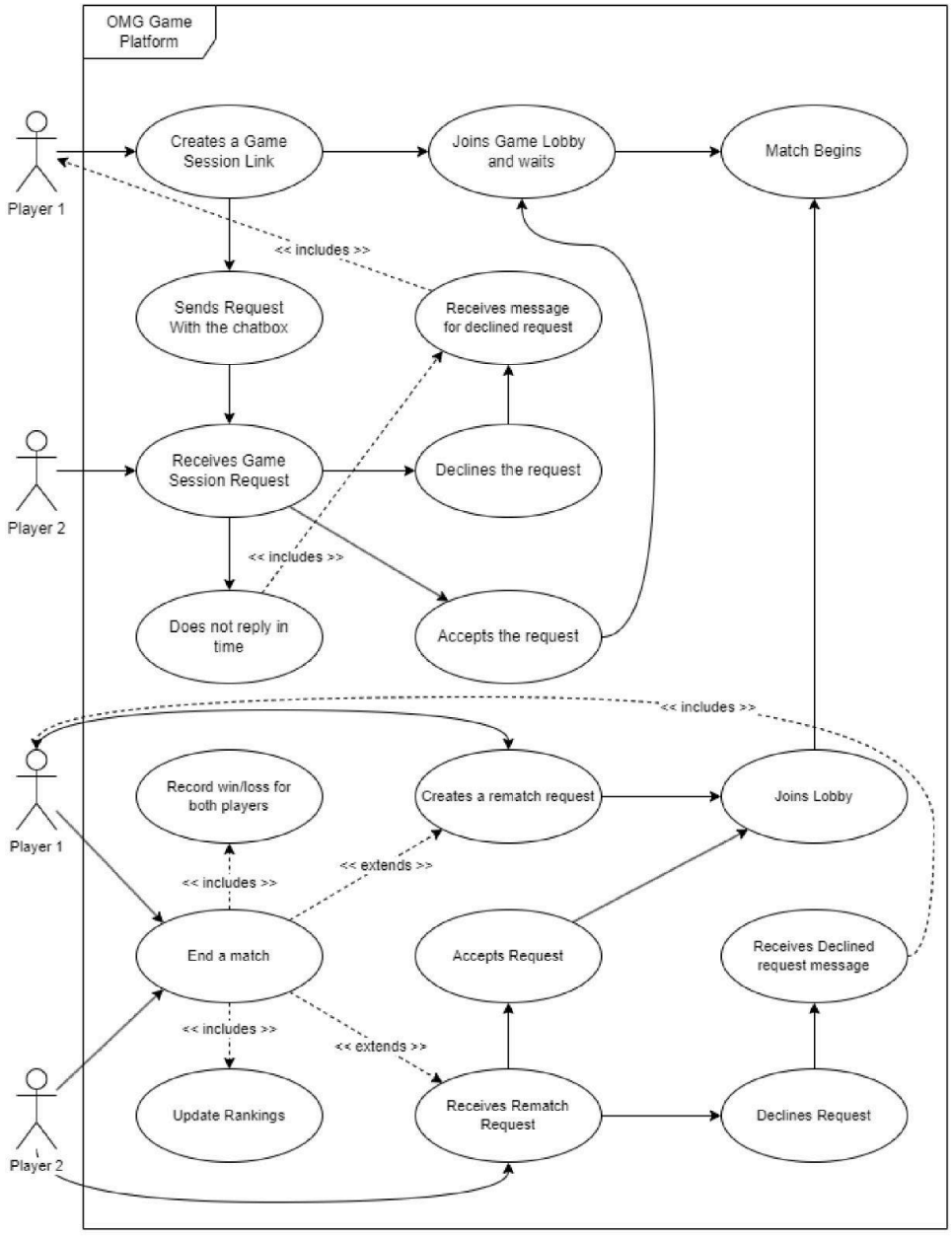
- Change `Check for win` and `Check for tie` from `<<extend>>` to `<<include>>` within `Move piece`.
- Ensure that `Resign` directly results in `Announce win`, rather than extending it.
- Add a `Validate move` use case to handle illegal moves.
- Separate system actions (automatic checks) from player-initiated ones for better clarity.
- Consider including a `Start Game` use case to indicate initial setup.

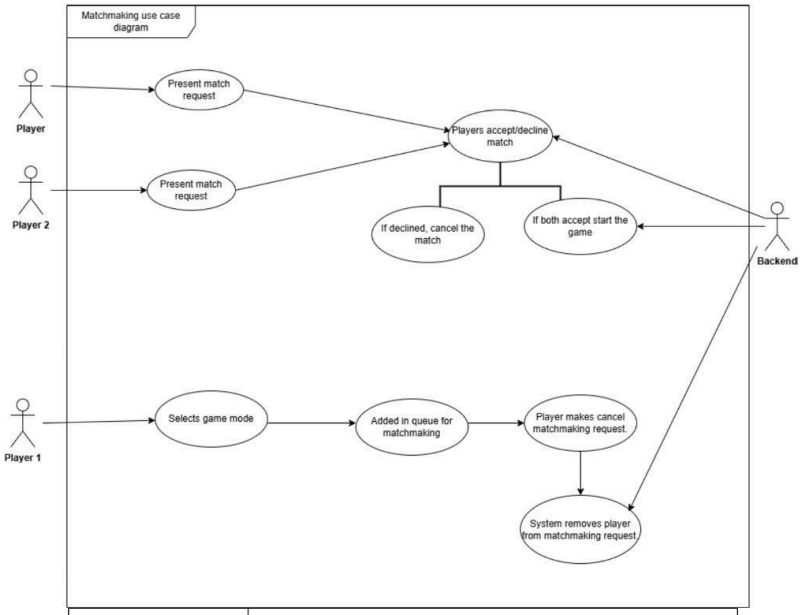
This is a **well-structured use case diagram** with clear relationships, but minor refinements could improve clarity and correctness.

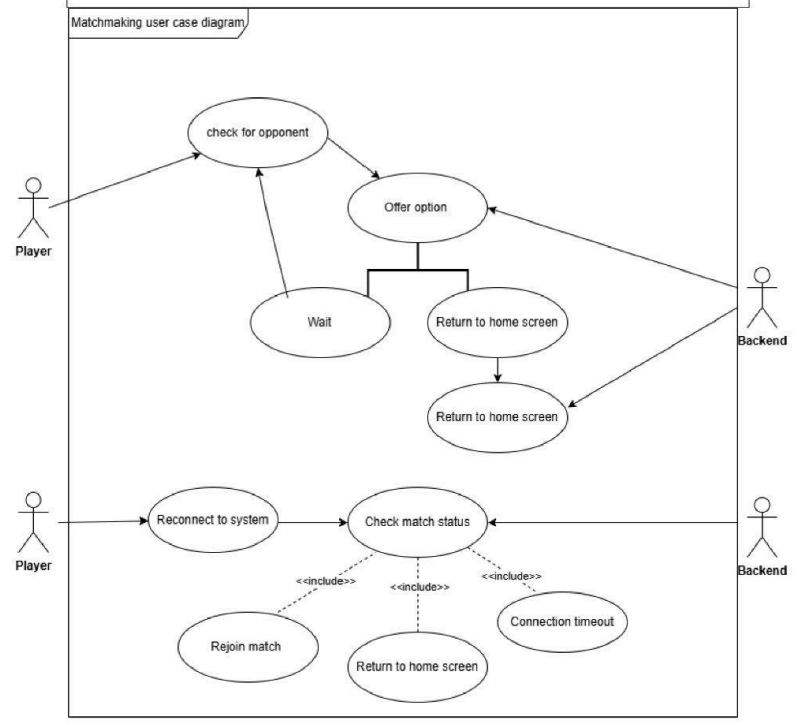
Leaderboard design

Matchmaking Use Case Diagrams



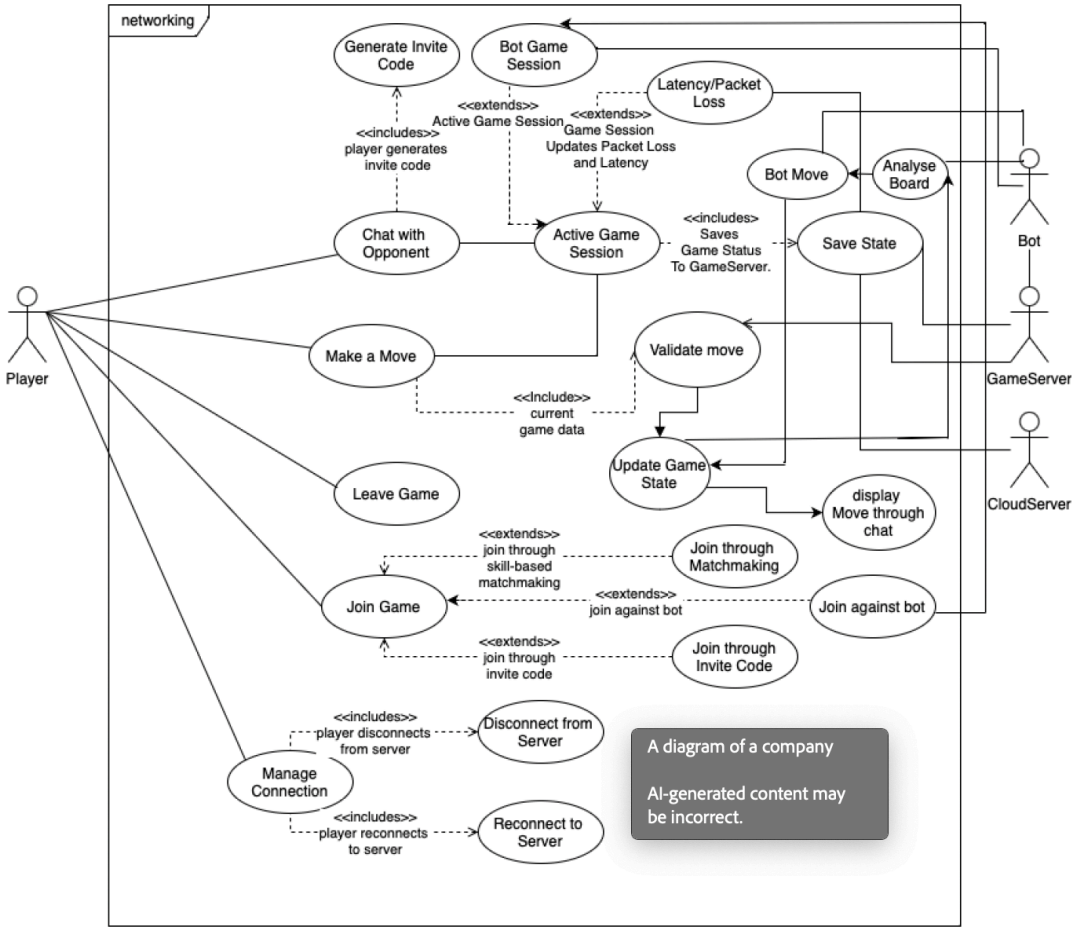

OMG Game Platform
Player 1
Creates a Game Session Link
Joins Game Lobby and waits
Match Begins
<< includes >>
Sends Request With the chatbox
Receives message for declined request
Player 2
Receives Game Session Request
Declines the request
<< includes >>
Does not reply in time
Accepts the request
<< includes >>
Player 1
Record win/loss for both players
Creates a rematch request
Joins Lobby
<< includes >>
<< extends >>
End a match
Accepts Request
Receives Declined request message
<< includes >>
<< extends >>
Update Rankings
Receives Rematch Request
Declines Request
Player 2


Matchmaking use case diagram
Player
Present match request
Player 2
Present match request
Players accept/decline match
If declined, cancel the match
If both accept start the game
Backend
Player 1
Selects game mode
Added in queue for matchmaking
Player makes cancel matchmaking request
System removes player from matchmaking request

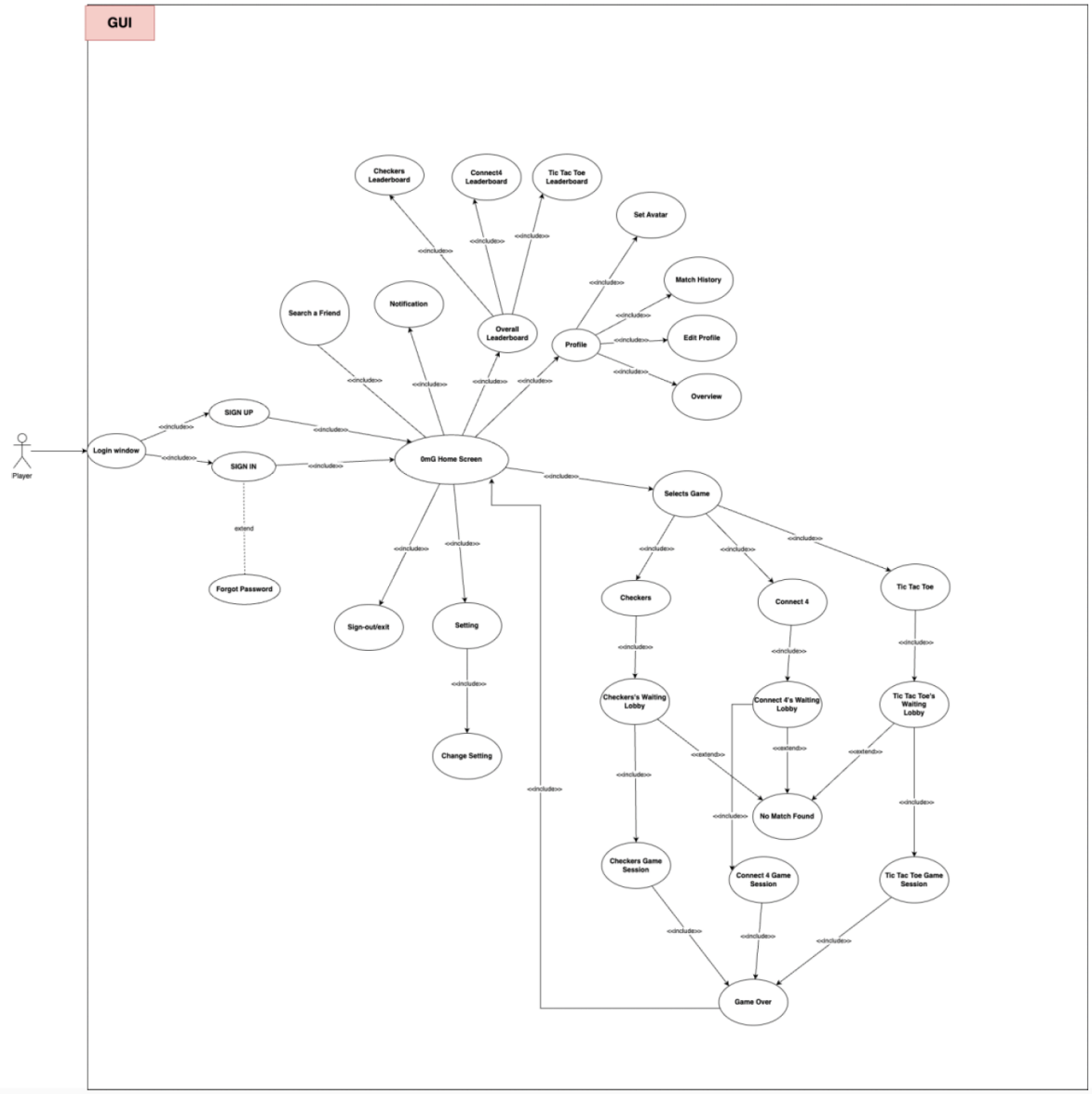

Matchmaking user case diagram
Player
check for opponent
Offer option
Wait
Return to home screen
Backend
Return to home screen
Player
Reconnect to system
Check match status
Backend
<<include>>
<<include>>
<<include>>
Rejoin match
Return to home screen
Connection timeout

Networking design

Use Case Diagram:



GUI designs


GUI
Checkers Leaderboard
Connect4 Leaderboard
Tic Tac Toe Leaderboard
Set Avatar
Match History
Search a Friend
Notification
Overall Leaderboard
Profile
Edit Profile
Overview
SIGN UP
Login window
SIGN IN
Player
0mG Home Screen
Selects Game
extend
Forgot Password
Sign-out/exit
Setting
Change Setting
Checkers
Connect 4
Tic Tac Toe
Checkers's Waiting Lobby
Connect 4's Waiting Lobby
Tic Tac Toe's Waiting Lobby
No Match Found
Checkers Game Session
Connect 4 Game Session
Tic Tac Toe Game Session
Game Over
«include»
«extend»

Use Case Descriptions

Networking Use Case Descriptions

Use case: Play a game with friends

Primary Actor: Player

Goal in context: To allow the player to connect to an online session with an online friend, and play games in real time.

Preconditions:

1. The player has an account on the online board game platform
2. The player's friends also have accounts on the platform
3. All players have a stable internet

connection **Trigger**: The player logs into the

system, then presses play. **Scenario**:

1. The player either:
    1. Creates a new game session and invites friends by sending them a unique game code
    2. Joins an existing session with other people already there by entering a game code.
2. The player sends invitations to their overseas friends via the platform's invite system.
3. The player waits for their friends to accept the invitation and join the game session
4. Once all the friends have joined, the player starts the game.
5. The player takes turns with their friends depending on the game. This game is updated in real time.
6. The player uses the platform's chat to interact with friends during the game.
7. The game concludes when a player wins or the game reaches its end condition.

8. The player can choose to:
    1. Exit the game session and return to main menu
    2. Start a rematch with the same group of friends.
9. The game session is closed, and all players are returned to main menu
10. The games results are saved to all players' profiles.

Exceptions:

1. If a friend declines an invitation, the player can either invite a new friend or start the game with remaining players
2. If a player loses connection, the game pauses and allows the player to reconnect within a minute and 30 second window.
3. If a player leaves the game mid-session, the platform allows the remaining players to continue or end.

Priority: High

When available: First iteration

Frequency of Use: Many times per day, for many users.
Channel to Actor: Player sends a request through the game.

Secondary Actors: Game Server

Open issues:

- What can be done to prevent spam?

Feedback:

1. **Preconditions:**
    - Consider adding a precondition that the game must support multiplayer functionality. Not all games may allow multiple players, so this should be clarified.
    - Suggestion: Add a precondition that the player and friends are online at the same time.
2. **Scenario:**
    - Step 2: Clarify how the "overseas friends" aspect is relevant. Is there a specific challenge with overseas connections (e.g., latency)? If so, this should be addressed in the exceptions or preconditions.

- Step 5: Specify how the game is updated in real time. Is this through the server? If so, mention the server's role in synchronization.
- Step 6: Consider adding more detail about the chat system. For example, is there a character limit? Can players send emojis or other media?

3. **Exceptions:**
   - Exception 2: Clarify what happens if the player cannot reconnect within the 1:30 window. Does the game end, or do other players continue?
   - Exception 3: Add an exception for server crashes or maintenance, which could disrupt the game session.
4. **Open Issues:**
   - The issue of spam prevention is important but vague. Suggest adding more detail: Are you concerned about spam in chat, invitations, or both? Consider adding a rate-limiting mechanism or CAPTCHA for invitations.

Use case: Reconnect

Iteration: 1

Primary Actor: The Game's player

Goal in context: When a player disconnects from the game via power outage, network disconnection, or other unexpected interruptions, there needs to be a way for the player to reconnect to the game that
they were playing without hindering the other player too much.

Preconditions: Game has to be in progress when they disconnect (can't reconnect before the game starts and after the game ends) and the player needs to rejoin in under a minute and 30 seconds.

Trigger: Player leaves the game via power outage, network disconnection, or other unexpected interruptions

Scenario:

1. Player is playing one of our amazing games and there is an unexpected disconnection
2. Player understands that they have 1:30 to get back on
3. Player resolves the reason for the unexpected disconnection
4. Player is able to rejoin if issue was resolved in the time limit
5. Player continues to enjoy their game

Post conditions: Player needs to be reconnected for at least 10 seconds before the game will resume. This will ensure the stability of the connection.

Exceptions:

1. Player doesn't join back in time

Priority: High priority as interruptions happen all the time. Lack of reconnection will hinder player experience

When available: When the user is disconnected from their game because of an unexpected interruption

Frequency of use: Somewhat frequent. Unexpected interruptions are quite common, and It's important to have something to deal with them

Channel to actor: Player must reopen the game and the connection will be reestablished

Secondary actors: Server

Channel to secondary actors: Internet connection

Open issues: *N/A*

Feedback:

1. **Preconditions:**
    - Consider adding a precondition that the game session must support reconnection. Some games may not allow reconnection after a disconnect.
2. **Scenario:**
    - Step 2: Clarify how the player is informed about the 1:30 window. Is there a notification or timer displayed on the screen?
    - Step 4: Add more detail about how the reconnection process works. Does the player need to re-enter the game code, or is it automatic?
3. **Exceptions:**
    - Add an exception for cases where the server itself is down, preventing reconnection.
4. **Post conditions:**
    - Clarify what happens if the player reconnects but the game has already ended. Are they informed of the result, or is the game abandoned?

Use case: Leave game

Iteration: 1

Primary Actor: Player

Goal in context: When the Player finishes a match or decides that they need to do something else at the moment they need an option to leave the game. Penalties maybe be applied to the player for leaving depending on how frequent the player leaves their matches. Ideally the player should be finishing all of their games, but that is not always going to be the case.

Preconditions: Player needs to be in a game

Trigger: At any point of the game player can press the leave button

Scenario:

1. Game has ended, and they need to leave the game to go back to the main page. (No penalty is applied)
2. Game is currently in progress and Player decides that they need to leave, but its only their first time leaving a game this week (Player gets a warning but no penalty)
3. Game is currently in progress and Player decides that they need to leave, but they have left more than 3 times this week (Player gets a penalty (can't play for 1 day))

4. If a player leave the game, the entire game ends and the other player will need to leave as well

Post conditions: Player must be returned to the main page of our program, so they can choose whether to play again or close the program entirely.

Exceptions:

1. If player has an unexpected disconnection that causes them to leave it will not count to the warnings the player has and will not cause a penalty

2. If the GUI button doesn't work player can leave by closing the game

Priority: High priority as we don't want to force our players to play our game. They need an option to leave when they need.

When available: Should be available at any point once the player is in a game

Frequency of use: Very frequent. Players will be leaving games all the time

Channel to actor: Button press through GUI

Secondary actors: N/A

Channel to secondary actors: N/A

Open issues: N/A

Feedback:

1. **Scenario:**
    - Step 4: Clarify what happens to the other player if the game ends abruptly. Are they informed of the reason for the game ending?
    - Consider adding a step where the player is asked to confirm their decision to leave, especially if penalties are involved.
2. **Exceptions:**
    - Exception 1: Clarify how the system distinguishes between an unexpected disconnection and a voluntary leave. Is this based on network diagnostics?
3. **Post conditions:**
    - Add a post-condition that the player's profile is updated to reflect the leave (e.g., a record of the leave is logged).

Use Case: Text Chat System During Game Session

Iteration: 1

Primary Actor: Player

Goal In Context: Allow players to send and receive text messages in real time without impacting the flow of the game through game network.

Preconditions:

1. The player has an account on the online board game platform
2. The player's friends or opponents also have accounts on the platform.
3. The players must be in a gaming session together that supports in-game chat feature.

Trigger: The player writes a message and sends it through the game chat interface.

Scenario:

1. A player opens chat input window in the game.
2. The player types a message and presses the "send" button.
3. The message is sent to game server network.
4. The server sends the message to all players in game session.
5. Each player's device updates their chat logs and displays the message in their chat window.

Exceptions:

1. If the player's internet connection is slow, the messages delivery may be delayed.
2. If the server is dysfunctional or players are disconnecting, the messages may be lost and not delivered via server network.

Priority: High, as text chat is important for player interaction and experience.

When Available: First increment.

Frequency Of Use: Frequently between multiple users.

Channel To Actor: Text messages are sent and received through in-game chat interface, which interacts with game server to display messages.

Secondary Actors: Game Server, Network Infrastructure

Channels To Secondary Actors:

- **Game Server**: Basic API calls to send and receive chat messages.
- **Network Infrastructure**: Handles player's connectivity and standard internet connection between device and the game server message transfer.

Open Issues:

1. How should the system handle messages if a player disconnects and reconnects? Should missed messages be stored and delivered when players reconnect?

Feedback:

1. **Preconditions:**
   - Consider adding a precondition that the chat feature must be enabled by the game session. Some games may not support chat.

2. **Scenario:**
    - Step 5: Clarify how long chat logs are stored. Are they saved only for the duration of the game, or are they stored permanently in the player's profile?
3. **Exceptions:**
    - Add an exception for cases where a player is muted or blocked by another player. How does the system handle this?
4. **Open Issues:**
    - The issue of missed messages is important. Suggest implementing a message queue that stores messages for a short period (e.g., 5 minutes) and delivers them upon reconnection.

Use case: Updating Moved Pieces

Iteration: 1

Primary Actor: Player

Goal in context: Ensure that moved game pieces are updated and synchronized in real time between the players and the server

Preconditions:

1. The player is connected to game server.
2. The game session is active, and players are synchronized.
3. The player has the right to move a piece based on game rules.

Trigger: The player interacts with game piece and moves it

Scenario:

1. A player selects a piece to move
2. The server validates the move based on game rules
3. The move is sent to server
4. Server verifies and update game state
5. Server broadcasts the updated moves to all connected players

6. Both interface for players updates and reflect the moves made
7. The updated game board is displayed on all players device

Post conditions:

1. The game state is updated and synchronized across all players
2. The move is tracked
3. All players game board shows the updated move visually

Exceptions:

- Invalid Move: If the move is not valid, then the system notifies the player and reject the action
- Ping latency: there is a delay in transmitting real time data to the server
- Disconnection: Player disconnects mid-game and the server handles reconnection

Priority: High, because real-time synchronization is important for fair game play

When available: First iteration

Frequency of Use: Continuously during gameplay

Channel to Actor: The player moves a piece through game interface, then it sends an update to the game server for synchronization.

Secondary Actors: Game Server

Channels To Secondary Actors:

- **Game Server**: performs move validation, state updates, and broadcasting game state changes

Open issues: How should system handle conflicts if both players try to move pieces simultaneously?

Feedback:

1. **Scenario:**
   - Step 2: Clarify how the server validates the move. Does it check against game rules stored on the server, or does it rely on the client's input?

    - Step 5: Add more detail about how the server broadcasts the updated moves. Is this done through a push notification system, or do clients poll the server for updates?
2. **Exceptions:**
    - Add an exception for cases where the server fails to validate a move due to a bug or error. How does the system recover from this?
3. **Open Issues:**
    - The issue of simultaneous moves is critical. Suggest implementing a move queue where moves are processed in the order they are received, and conflicts are resolved based on game rules.

Use Case: Play Against AI Bot

Primary Actor: Developer

Goal in Context: To allow players to challenge the developer's AI bot on the server for skill improvement.

Preconditions:

- The developer has deployed the AI bot on the game server.
- The player has an account on the online board game platform.
- The player has a stable internet connection.

Trigger:

The player logs into the system, selects "Play Against AI," and starts a match.

Scenario:

1. The player selects the option to play against the AI bot.
2. The game server assigns the AI bot as the opponent.
3. The player starts the game and takes turns against the AI.
4. The AI bot makes moves based on its programmed strategy.

5. The player can adjust the AI difficulty level before or during the game.
6. The game updates in real time, ensuring smooth gameplay.
7. The player can use an optional analysis tool to review their moves and learn from the AI.
8. The game concludes when a player wins, the AI wins, or an end condition is met.
9. The player can choose to:
    - Start a rematch against the AI.
    - Return to the main menu.
10. The game results and AI performance metrics are saved to the player's profile for review.

Exceptions:

- If the AI bot fails to respond within a set time, the game pauses and attempts to reconnect.
- If the player disconnects, they are given a chance to rejoin within a minute and 30 seconds.
- If the AI experiences a server error, the player can restart the session or report an issue.

Priority: High

When Available:: First iteration

Frequency of Use:: Many times per day, for many users.

Channel to Actor:: The player initiates a game request through the platform.

Secondary Actors:: Game Server, AI Bot

Open Issues:: AI Difficulty Scaling – Should the AI have multiple difficulty levels, and how should they be adjusted dynamically during a match?

Feedback:

1. **Preconditions:**
    - Consider adding a precondition that the AI bot must be online and available for play.
2. **Scenario:**
    - Step 5: Clarify how the player adjusts the AI difficulty level. Is this done through a menu, or can it be done mid-game?

    - Step 7: Add more detail about the optional analysis tool. What kind of analysis is provided? Is it a replay of the game, or does it offer strategic advice?
3. **Exceptions:**
    - Add an exception for cases where the AI bot makes an invalid move due to a bug. How does the system handle this?
4. **Open Issues:**
    - The issue of AI difficulty scaling is important. Suggest implementing a dynamic difficulty system where the AI adjusts its strategy based on the player's performance during the match.

General Feedback:

- **Consistency:** Ensure that all use cases follow a consistent structure (e.g., preconditions, triggers, scenarios, exceptions). Some use cases are more detailed than others, which can lead to confusion.
- **Technical Details:** Consider adding more technical details where relevant, such as how the server handles data synchronization, how messages are queued, or how the AI bot processes moves.
- **User Experience:** Think about the user experience in each use case. For example, how are players informed of penalties, reconnection status, or game updates? Adding more detail here can improve clarity.
- **Error Handling:** Ensure that all use cases have robust error handling, especially for network-related issues like disconnections or server errors.

UI Designs