CS262 Final Project Engineering Notebook

– A Distributed Multiplayer System for Five-in-a-Row

Hanying Feng and Vera Yang

**Link to code repo:** <https://github.com/verayang01/CS262-final-project>

## Introduction

Five-in-a-Row, also called Gomoku, is a classic strategy board game where two players compete to place five of their stones–black or white–in a consecutive row (horizontally, vertically, or diagonally) on a 19x19 grid. This project builds a distributed multiplayer system for playing Five-in-a-Row online, featuring the following functionalities:

* User account creation and authentication (login/signup)
* Real-time gameplay with forced game rules, turn timers, and smooth disconnection handling
* Matching room for player searching, game invites, and private matching
* Live game viewer mode
* Game history and historic game replays
* Player statistics maintenance and leaderboard ranking
* User account management
* Incentive-based game credits and ranking system

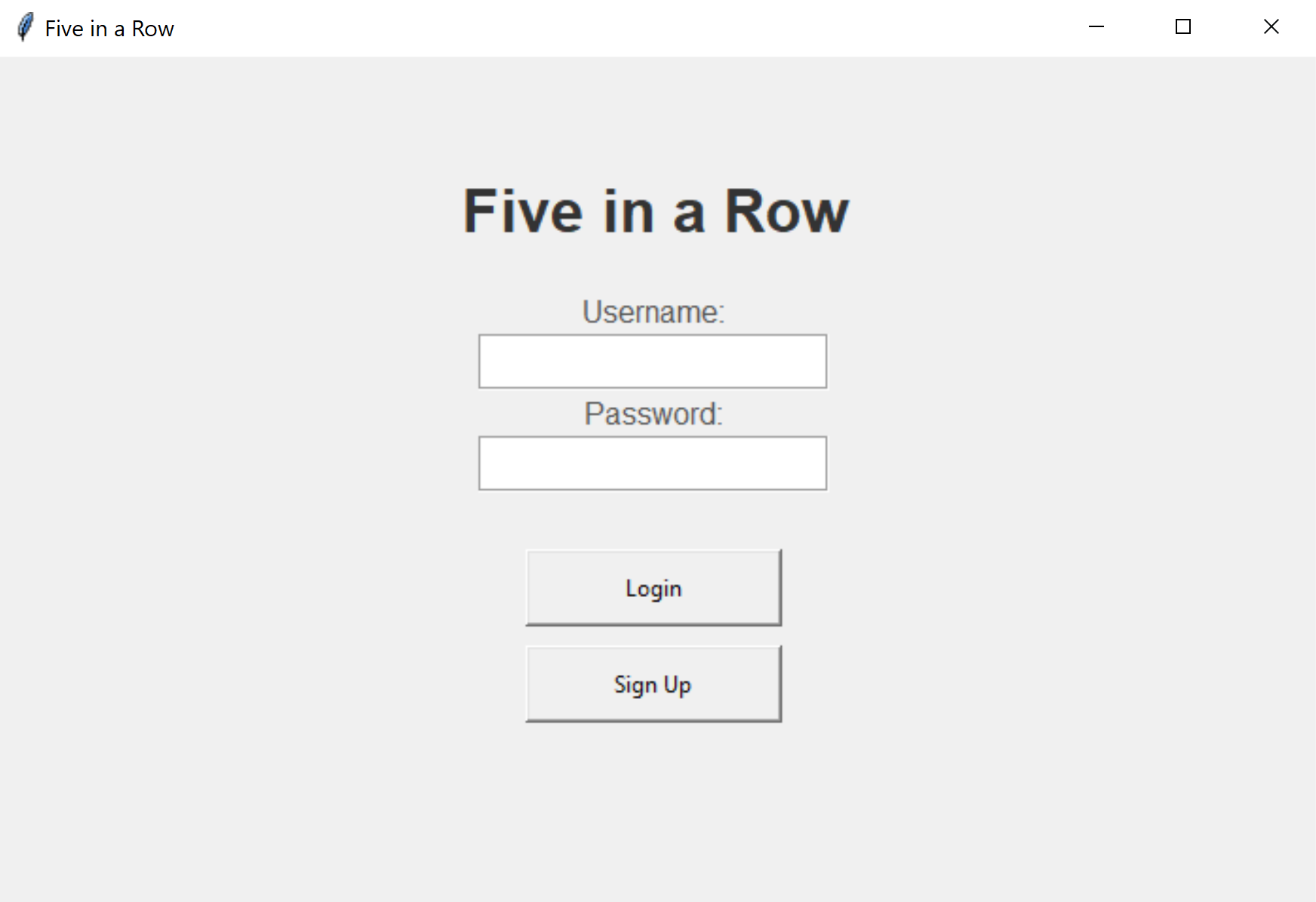
We developed the entire system using Python and designed and implemented the client GUI using TKinter. Throughout this project, we utilized knowledge learned from class, such as TCP socket connections, client and server communication, wire protocols, and error handling. We have found that the real-time nature of the online pairing and game playing posed new challenges to the development of this application, and we found it rewarding to find solutions to these challenges by building up the application step-by-step. In this document, we summarize our design specifications, implementation details, testing, and our thought processes to present how we put these building blocks together.

## Design Specification:

### Authentication

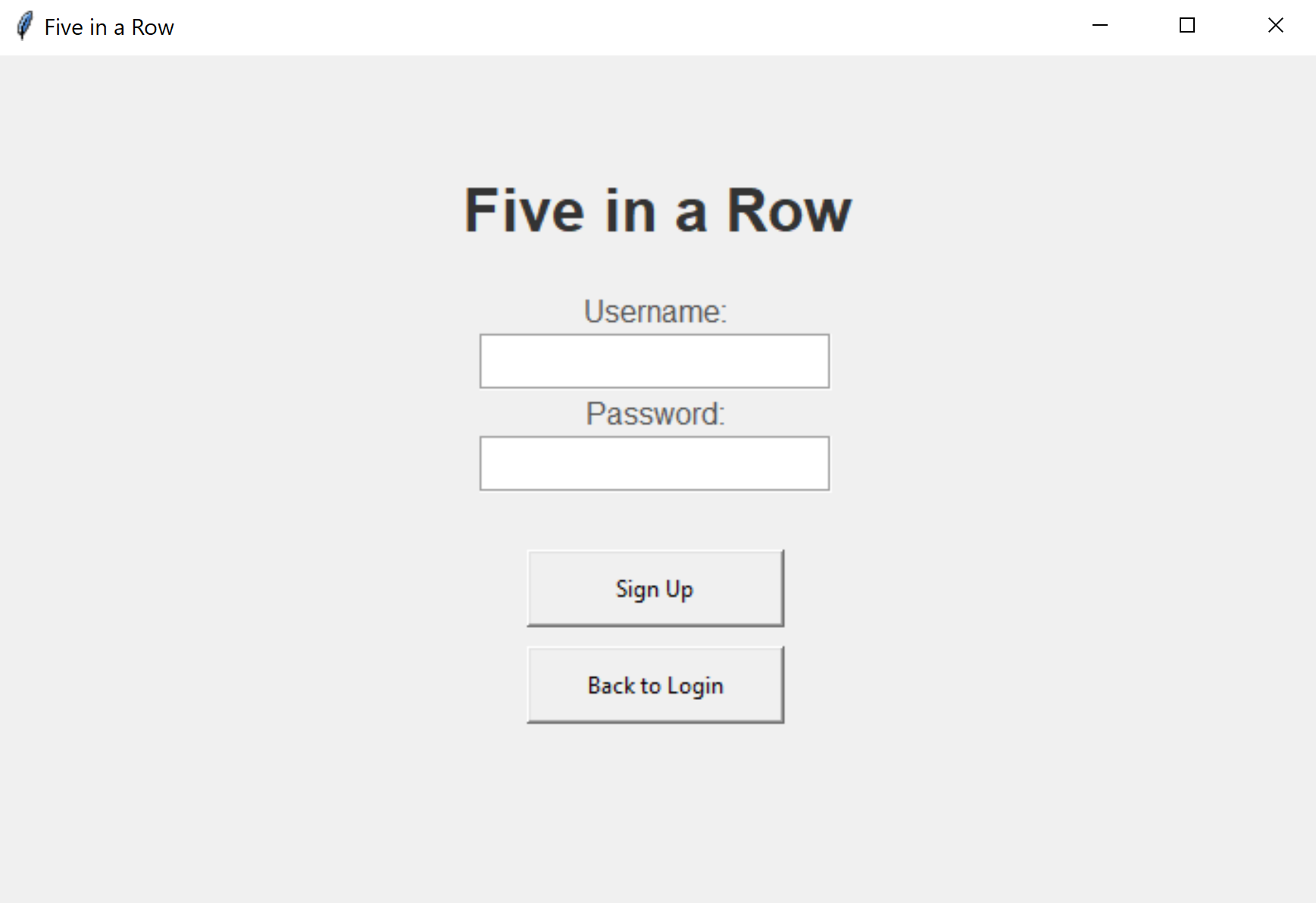
1. **Functionality Details and User Logic**

When the user first runs the code, a login page will appear. The login page contains two fields that allow the user to enter a username and password. When the user clicks the “Login” button, the entered information will be sent to the server for authentication, and the user can log in successfully if the credentials are correct. The user needs to sign up before logging into an account with a new username. Each user can only be logged in once, and they must log out before logging in from another client.



(Login)

When the user clicks the “Sign Up” button, they will be directed to the sign-up page. This page also contains two fields that allow the user to enter a username and password. The username must not duplicate any existing signed-up usernames, and both the username and password must not be empty or contain any spaces or newline characters. After entering a valid username and password, clicking the “Sign Up” button will create the account and redirect the user to the login page so they can log in and play the game. There is also a “Back to Login” button at the bottom that allows the user to return to the login page without creating a new account.



(Sign Up)

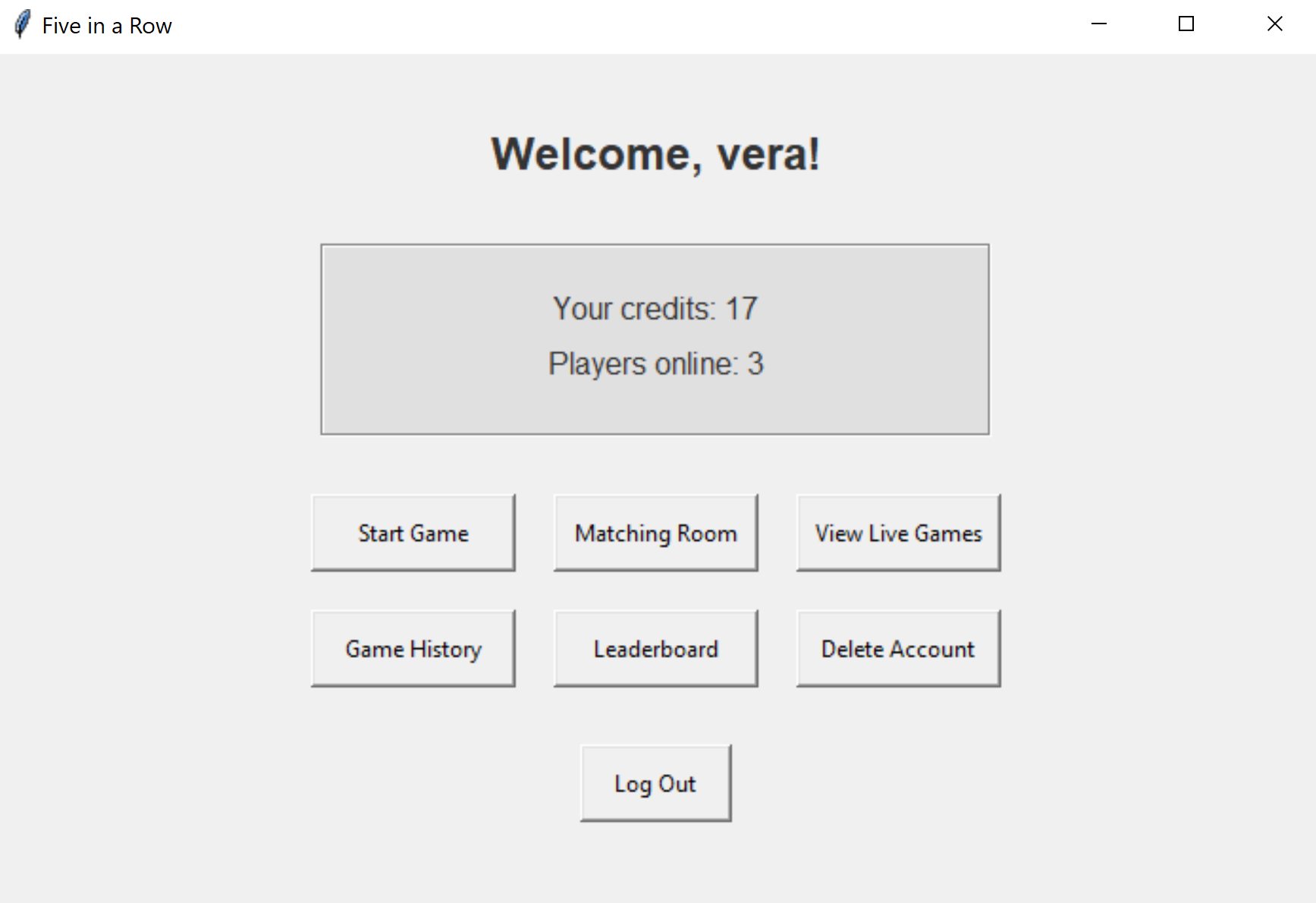
1. **Code Implementation**

* Client-side implementation
  + Class AuthUI: Add an authentication UI class for login and sign-up page.
    - \_\_init\_\_(self, root, client): initialize the authentication UI and call the function to draw the login page.
    - show\_login\_form(self): displays the login form with username/password fields and login/sign up buttons.
    - show\_signup\_form(self): displays the login form with username/password fields and sign up/back to logins buttons.
    - login(self): checks if the username and password are not empty and sends a LOGIN\_REQUEST message to the server to validate the inputs.
    - attempt\_signup(self): checks if the username and password are not empty and do not contain any space/newline character. Sends a SIGNUP\_REQUEST message to the server if the inputs are appropriate.
  + Class GameClient: The game client class for handling responses from the server. Add methods to handle LOGIN\_RESPONSE and SIGNUP\_RESPONSE. Add a method to show authentication UI.
    - handle\_server\_message(self, message):
      * Handles LOGIN\_RESPONSE by calling show\_home\_ui to redirect to the home page.
      * Handles SIGNUP\_RESPONSE by calling show\_auth\_ui to redirect to the login page for the user to login.
    - show\_auth\_ui(self): display the authentication UI.
* Server-side implementation
  + Class GameServer: Add message handling for client messages sent from authentication UI.
    - \_process\_message(self, message, client\_socket):
      * Handles SIGNUP\_REQUEST by calling handle\_message from class UserManager class.
      * Handles LOGIN\_REQUEST by calling handle\_message from class UserManager class. Send LOGIN\_RESPONSE message to player if login successfully.
  + Class UserManager: Add methods to handle SIGNUP\_REQUEST and LOGIN\_REQUEST.
    - handle\_message(self, message):
      * Handles SIGNUP\_REQUEST by calling \_handle\_signup, which checks that the username is not duplicated and saves the username and hashed password to the server database. It returns a SIGNUP\_RESPONSE message indicating a successful sign-up.
      * Handles LOGIN\_REQUEST by calling \_handle\_login. It checks that the user is not attempting to log in repeatedly, verifies that the username has already signed up, and confirms that the entered password is correct. The server then adds the username to the logged\_in\_users set and returns a LOGIN\_RESPONSE message containing basic information about the current user, including credits, wins, and losses.
    - \_hash\_password(self, password): securely hashes the password using SHA-256 algorithm. Return the hashed password.
  + Class Database: Add methods to save and retrieve user data from the server database.
    - save\_user(self, user): saves the user information to the database.
    - get\_user(self, username): retrieves the user information from the database.
    - \_load\_users(self): load all user data from the users file.
    - \_save\_users(self, users): save all user data to the users file.

1. **Development Process and Challenges**

* We initially did not have a sign-up function, and users could create a new account simply by entering a new username and clicking the “Login” button on the login page. Later, we decided it would be clearer and more organized to require users to sign up before logging in. As a result, we added a sign-up page to allow users to create new accounts.
* In the beginning, the only restrictions on the username and password were that the username could not be duplicated (i.e., it must not already exist in the database) and could not be empty. While developing other functionalities, we introduced the newline character as a delimiter for messages sent between the client and server. To prevent parsing errors, we added further restrictions requiring that the username and password must not contain spaces or newline characters.
* At first, we did not restrict repeated logins for the same username. However, we realized that it didn’t make sense for the same account to be logged in at multiple locations and playing games with different players simultaneously. Therefore, we introduced a new logged\_in\_users set to track currently online users and ensure that each username can only be logged in at one time.

### Homepage



1. **Functionality Details and User Logic**

After a successful authentication and logging in, the user enters the home page. The homepage shows a welcome message at the top. The middle box shows the game credits of the logged in user and the number of players online that are both updated in real time. The homepage provides seven main options for the user:

1. Start Game: The quick start option that allows the user to join the matchmaking queue and start a game with the next online player in the queue.
2. Matching Room: The user can join the matching room and pair up with other players who also wish to find a match. The user can also send and receive game invites.
3. View Live Games: Watch ongoing games between other players.
4. Game History: Review past game history and credit changes and replay historic games.
5. Leaderboard: View the top 100 players ranked by game credits.
6. Delete Account: Permanently remove the account and associated data.
7. Log Out: Log out of current account.
8. **Code Implementation**

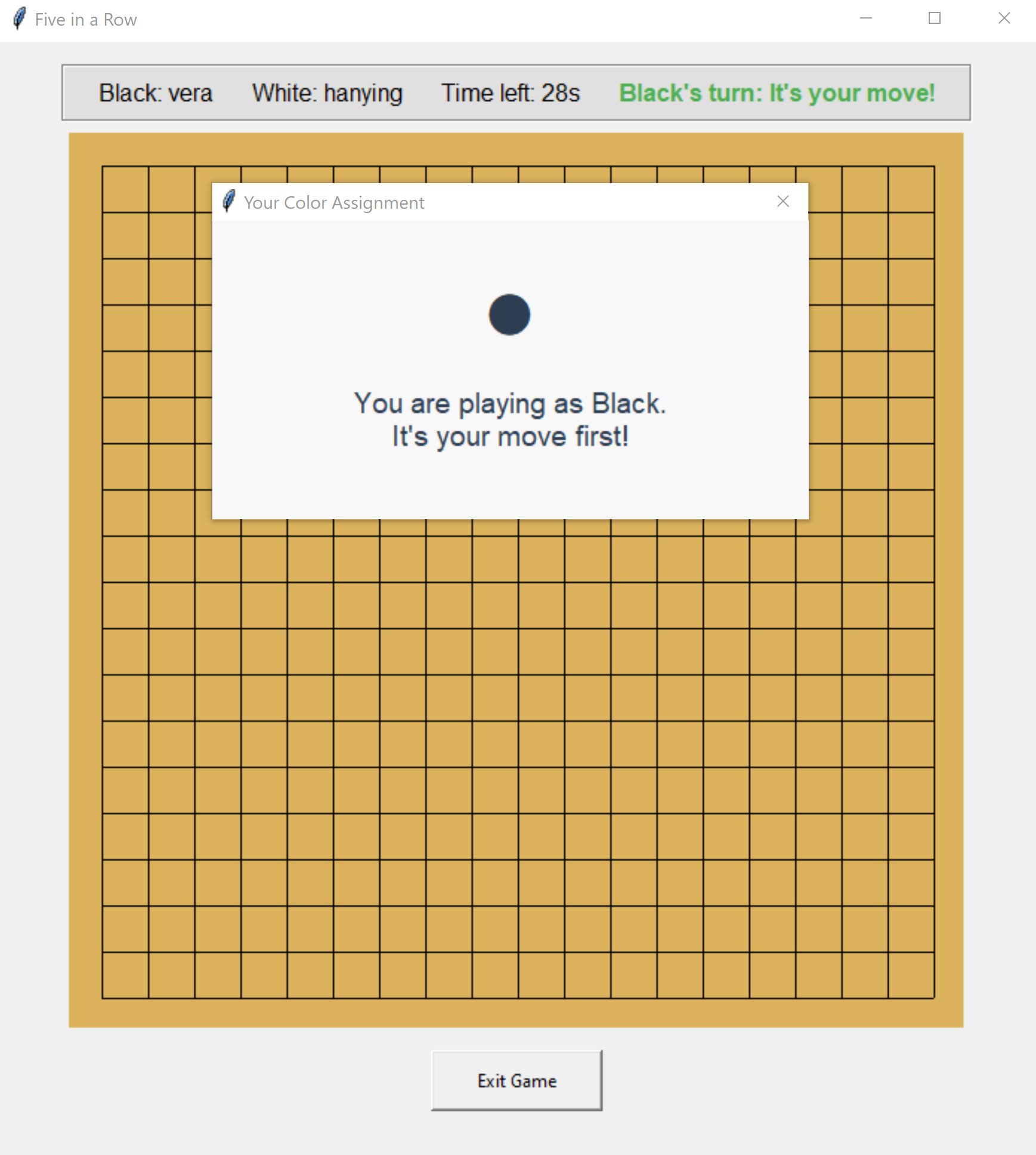
* Client-side implementation
  + Class HomeUI: Add a separate UI class for the homepage.
    - \_\_init\_\_(self, root, client): Draw the frames and buttons and show the welcome message. “Credits” and “Players Online” are default to show “Loading…” if data retrieval is not successful.
    - update\_stats\_ui(self,data): update credits and players online with given data.
    - start\_game(self): show the waiting UI and send a QUEUE\_REQUEST message to the server with action=join.
    - show\_live\_games(self): show the live games viewer UI and send a GET\_LIVE\_GAMES\_REQUEST message to the server.
    - show\_history(self): show the history UI and send a GET\_HISTORY\_REQUEST message to the server.
    - show\_leaderboard(self): show the leaderboard UI and send a GET\_LEADERBOARD\_REQUEST message to the server.
    - show\_matching\_room(self): show the matching room UI.
    - logout(self): send a LOGOUT message to the server, clear the current client state, and return to the authentication UI.
    - delete\_account(self): pop up a message box to ask the user to confirm deletion. If “yes”, stop polling homepage stats and send an ACCOUNT\_DELETE\_REQUEST message to the server.
    - start\_polling(self): call the poll\_stats function to start polling homepage statistics for credits and players online.
    - stop\_polling(self): stop polling homepage statistics.
    - poll\_stats(self): send a GET\_STATS\_REQUEST to the server every 0.4 seconds.
  + Class GameClient: The game client class for handling responses from the server. Add methods to handle GET\_STATS\_RESPONSE and MATCH\_FOUND. Add a method to show home UI.
    - handle\_server\_message(self, message):
      * Handles GET\_STATS\_RESPONSE by calling update\_stats\_ui from class HomeUI.
      * Handles MATCH\_FOUND by calling start\_game and calling handle\_match\_response from MatchingRoomUI if in the matching room.
    - show\_home\_ui(self): display the home screen.
* Server-side implementation
  + Class GameServer: Add message handling for client messages sent from home UI.
    - \_process\_message(self, message, client\_socket):
      * Handle QUEUE\_REQUEST by calling handle\_message from Class MatchMaking. If the response is MATCH\_FOUND then create a game between two players and send a MATCH\_FOUND message to both players.
      * Handle GET\_LIVE\_GAMES\_REQUEST and GET\_HISTORY\_REQUEST by calling handle\_message from Class GameManager.
      * Handle GET\_LEADERBOARD\_REQUEST by calling get\_leaderboard from Class UserManager.
      * Handle LOGOUT, ACCOUNT\_DELETE\_REQUEST, and GET\_STATS\_REQUEST by calling handle\_message from class UserManager.
  + Class MatchMaking: Add methods to handle QUEUE\_REQUEST.
    - handle\_message(self, message): calls \_handle\_queue\_request.
    - \_handle\_queue\_request(self, message): If action=join, append the user to the matching queue. Call get\_queue\_size to get the current queue size. If queue size > 2, return a MATCH\_FOUND message, otherwise return a QUEUE\_RESPONSE message with status=waiting. If action=leave, remove the user from the queue and return a QUEUE\_RESPONSE message with status=left\_queue.
    - get\_queue\_size(self): returns current size of queue.
  + Class GameManager: Add methods to handle GET\_LIVE\_GAMES\_REQUEST and GET\_HISTORY\_REQUEST.
    - handle\_message(self, message):
      * Handles GET\_LIVE\_GAMES\_REQUEST by calling \_handle\_get\_live, which returns a message containing the ongoing live games saved in the server database.
      * Handles GET\_HISTORY\_REQUEST by calling \_handle\_get\_history, which returns a message containing the user game history saved in the server database.
  + Class UserManager: Add methods to handle GET\_LEADERBOARD\_REQUEST, LOGOUT, ACCOUNT\_DELETE\_REQUEST, and GET\_STATS\_REQUEST.
    - handle\_message(self, message):
      * Handles LOGOUT by calling \_handle\_logout, which discards the logged in user and returns a LOGOUT message.
      * Handles ACCOUNT\_DELETE\_REQUEST by calling \_handle\_delete\_account, which changes the user’s username to “account deleted” in all game histories, remove user from the user database, and returns an ACCOUNT\_DELETE\_RESPONSE message.
      * Handles GET\_STATS\_REQUEST by calling \_handle\_get\_stats, which returns a GET\_STATS\_RESPONSE message that contains the user’s game credits and the number of online players.
    - get\_leaderboard(self, limit): returns the list of top 100 users sorted by game credits in descending order.

1. **Development Process and Challenges**

* We started by sending a single request to the server to show the credits and number of online players, but then we realized that it would only make sense if these stats update automatically as they change. Thus, we added polling on the homepage to allow the numbers to update lively. For instance, the user will see the number of online players change as other players log in and out.
* At first we only used a single server class for handling all client requests, but soon we found it challenging to build all functionalities with a single class, so we divided the server into several classes by their roles, such as UserManager, GameManager, MatchMaking, GameServer, etc.
* We encountered errors when returning to the homepage and we found that the errors occurred because there were other polling processes called by other UIs that were still going on in the backend. Thus, we make sure that all other polling processes are stopped before leaving its associated page and returning to home.

### Start game

#### Game Playing and Live Game Handling



1. **Functionality Details and User Logic**

After pairing up with another player, the user enters the game immediately. The user will be randomly assigned a stone color–black or white. In any case, the black stone plays first. The user will see a message indicating their stone color and if they move first or second. The top bar of the game page shows the stone colors and the associated users. It also shows the time left for the current turn. Players should finish a turn in 30 seconds. If they do not place a stone in 30 seconds, the system will randomly place a stone for them and switch turns. On the top right the user can also see who is playing the current turn. The user can place a stone by clicking any empty intersections on the 19\*19 board. The board will update lively.

The winner is the first player who achieves consecutive five stones horizontally, vertically, or diagonally. When a win is detected, the game ends and the user sees a game over message that indicates who the winner was. The message will also show the game credit changes for the user. During the game, the user can exit at any time by clicking “Exit Game” at the bottom. However, this will count the opponent as the winner. There is no draw. If the board is full and no player achieves consecutive five stones, the white stone wins because the black stone has a first move advantage.

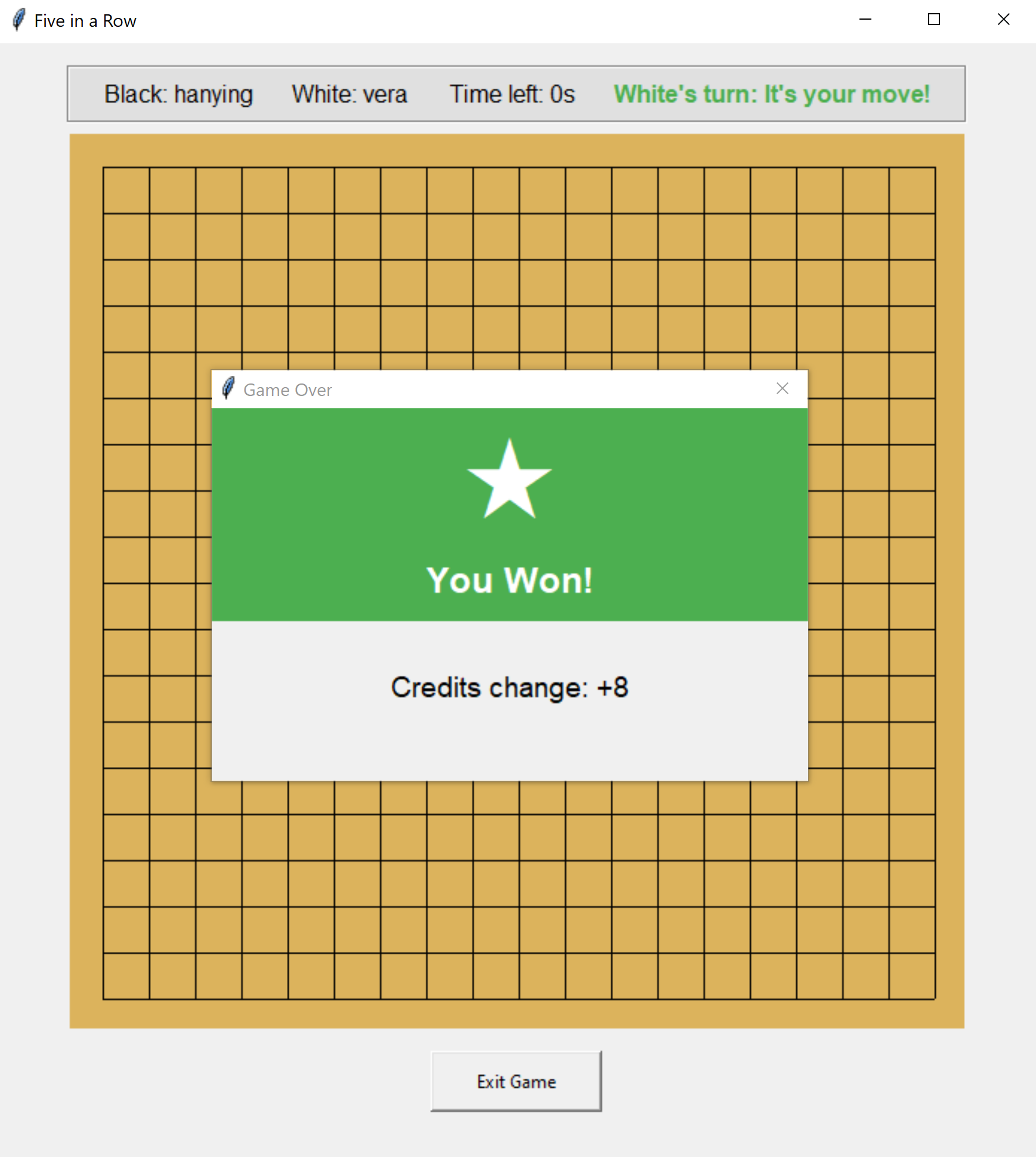
1. **Code Implementation**

* Client-side Implementation
  + Class WaitingUI: The UI for the waiting message when a user enters the “Start Game” matching queue and could not find another online user in the queue.
    - \_\_init\_\_(self, root, client): draw the waiting message box and show “Waiting for another player”. Draw a cancel button that calls cancel\_waiting on click.
    - animate\_dots(self): animation for waiting dots.
    - animate\_spinner(self): animation for loading icon.
    - cancel\_waiting(self): send a QUEUE\_REQUEST message to the server with action=leave.
  + Class GameUI: The main game UI that holds a live game between two players.
    - \_init\_\_(self, root, client, player1, player2, view\_mode, game\_id, replay\_mode, winner): initialize the game UI with player information and game settings. Draw the game board and set widgets. If not replay mode, start polling game state.
    - show\_color\_assignment(self): displays a window indicating the player’s assigned color and the rule that black moves first.
    - create\_widgets(self, view\_mode, replay\_mode, winner): include player information labels. Show timer and turn labels in game mode and view mode. Show winner in replay mode. Add the “Exit Game” button for game mode.
    - draw\_board(self): draw the 19\*19 game board. Call draw\_stone to draw existing stones on board.
    - draw\_stone(self, row, col, color, move\_number): draw a stone on the board at the specified location. In replay mode, draw the move number on top of the stone to show the order of stone placement.
    - update\_game\_state(self, state, view, replay): update game UI based on the latest game state from the server. If in game mode, set turns such that only the current turn player can place a stone. In game mode and view mode, show player information, time left, and current turn information. Call draw\_board and draw\_stone to draw the game board. Add move numbers if replay mode.
    - start\_local\_timer(self): start the countdown timer for the current player’s turn.
    - tick(self): callback for timer countdown, updates every second.
    - stop\_local\_timer(self): stop the local time if running.
    - on\_click(self, event): handle mouse click events on the game board. Converts click coordinates to board position. Send a MAKE\_MOVE message to the server containing the row and column of stone placement.
    - start\_polling(self, game\_id): call poll\_game\_state to start polling.
    - stop\_polling(self): stop polling game state.
    - poll\_game\_state(self, game\_id): request the current game state from the server and schedule the next poll after 0.5 seconds. Send a GET\_GAME\_STATE message to the server with the game id.
    - show\_game\_over(self, winner, credits\_change): stop polling game state and display the game over dialogue with the result and credits change.
    - exit\_game(self): handle game exits before game is over. Send a PLAYER\_DISCONNECTED message to the server. Show the home UI.
  + Class GameClient: The game client class for handling responses from the server. Add methods to handle GAME\_STATE and GAME\_OVER. Add methods to show the waiting UI and start the game. Add methods to handle window close.
    - \_\_init\_\_(self, root): initialize game client and establish connection to the server.
    - connect(self): establish connection to the server.
    - send\_message(self, message): send a message to the server.
    - receive\_message(self): continuously receive messages from the server.
    - handle\_server\_message(self, message):
      * Handles GAME\_STATE by calling update\_game\_state from GameUI. If in live game viewer, call update\_game\_state from LiveGameViewerUI.
      * Handles GAME\_OVER by calling show\_game\_over from GameUI.
    - show\_waiting\_ui(self): display the waiting screen.
    - start\_game(self, black\_player, white\_player, game\_id): call \_clear\_current\_ui to clear the current UI. Create a GameUI instance with the player information.
    - \_clear\_current\_ui(self): cleans up the currently displayed UI.
    - on\_window\_close(self): send a PLAYER\_DISCONNECTED message to the server.
* Server-side Implementation
  + Class GameServer: Add message handling for client messages sent from waiting UI and game UI.
    - \_process\_message(self, message, client\_socket):
      * Handles QUEUE\_REQUEST by calling handle\_message from Class MatchMaking.
      * Handles MAKE\_MOVE by calling handle\_message from Class GameManager. If not game over, send a GAME\_STATE message to both players. If game over, send both GAME\_STATE and GAME\_OVER messages to both players.
      * Handles GET\_GAME\_STATE by calling handle\_message from Class GameManager.
      * Handles PLAYER\_DISCONNECTED by calling handle\_message from Class GameManager. Send a GAME\_OVER message to both players.
    - \_handle\_client(self, client\_socket): if a user disconnects from the server during a game, call GameManager to handle a PLAYER\_DISCONNECTED message. Send a GAME\_OVER message to the remaining player.
  + Class GameManager: Add methods to handle MAKE\_MOVE, GET\_GAME\_STATE, and PLAYER\_DISCONNECTED. Add methods to handle all game-related operations, including game creation, move validation, win detection, and game state persistence.
    - \_\_init\_\_(self, user\_manager): initialize the game manager with a user manager instance.
    - create\_game(self, player1, player2): create a new game between two players. Randomly generate a game id. Randomly assign stone colors to players. Initialize an empty game board. Record initial game state and call save\_live\_game from Database.
    - handle\_message(self, message):
      * Handles MAKE\_MOVE by calling \_handle\_make\_move.
      * Handles GET\_GAME\_STATE by loading live games from the live games database. Return a GAME\_STATE message.
      * Handles PLAYER\_DISCONNECTED by calling \_handle\_player\_disconnected.
    - \_handle\_make\_move(self, message): process a move and update game state. First, call \_is\_valid\_move to check if the move is legal. If not a legal move, return an ERROR message. Process the move on the board and call save\_live\_game to save the latest game state. Call \_check\_winner to check if this move resulted in a win. If the board is full and no winner is determined, the white stone wins. If the board is not full and no winner is determined, switch turns, update the turn timer, and return a GAME\_STATE message. If a winner is determined, call calculate\_credit\_change to calculate the credit changes for each player. Call update\_user\_stats from UserManager to update user game credits. Call save\_game\_history from Database to save the game, and call delete\_live\_game from Database to remove the game from the currently active games. Return a tuple of a GAME\_STATE message and a GAME\_OVER message.
    - \_is\_valid\_move(self, game, username, row, col): check if a move is legal.
    - \_check\_winner(self, game, row, col): check if the last move resulted in a win.
    - \_handle\_player\_disconnected(self, message): handle player disconnection by ending the game and awarding win to the remaining player. First, identify the remaining player as the winner and call calculate\_credit\_change to calculate credit changes. Call update\_user\_stats from UserManager to update user game credits. Call save\_game\_history from Database to save the game, and call delete\_live\_game from Database to remove the game from the currently active games. Return a GAME\_OVER message.
  + Class Database: Add methods to manage game history and live games data.
    - save\_game\_history(self, history): save a completed game to the database.
    - save\_live\_game(self, game\_id, game\_state): save the current state of an active game to the database.
    - delete\_live\_game(self, game\_id): remove a completed game from the active games database.

1. **Development Process and Challenges**

* The first problem we encountered was that after the user clicks “Start Game” and sees a waiting screen, they can click the button again and open another waiting screen, which did not make sense. Thus, we ensure that we clear the home UI before showing the waiting UI such that double waiting UIs will not appear.
* At first we found that we could not see the board changes even though the moves are recorded in the server's database. We found that it was because we were not updating the board on the UI end. Thus, we added polling in the game UI to continuously poll the board status. In this way, all changes to the board are reflected on both players’ ends immediately.
* We initially did not include a turn timer. However, we thought about the case when it takes forever for the opponent to place a stone, making the other player wait. To enhance the user experience, we included a 30 second countdown timer for each turn. The players are required to place a stone within the timeframe, or the system will place a stone for them at a random place. To implement this timer, we thought about two ways–countdown by the server or by the client. In order to ensure that both players have a consensus on the remaining time of the current turn, we decided to implement the countdown on the server’s end by adding a “time\_remaining” field in the game state. The clients will then poll the remaining time as part of a game state polling.
* For invalid moves, initially we designed pop-up windows for users when they are trying to make an invalid move, say, when it is not their turn or the move is out of boundary. However, we think it makes more sense if the invalid moves are silently disabled. For example, when a user randomly clicks on the board when it is not their turn, the system should ignore their actions instead of popping up error windows saying their moves are invalid.
* We initially only implemented the handling of a disconnected player if clicking “Exit Game”. However, there are cases when the user closes the application or unexpectedly loses connection with the server. Thus, to handle these cases, we added game over handling on window close or player disconnection. In all cases, if a player exits the game we will count the opponent as the winner.
* After the game ends, at first we pop the user back to the home screen immediately. However, we later thought that it would be better if the user can remain on the board to review the game if they would like to. Thus, we enhanced this feature by add an additional polling after the game ends, such that the final game board is shown and the user can then click “Exit Game” to return to the home screen.

#### Game Credit Calculation



1. **Functionality Details and User Logic**

The credit system is designed to reflect the competitiveness, difficulty, and efficiency of a game in a scalable and fair manner. When a game ends, the system adjusts both players’ credits based on the outcome. The winning player gains credits while the losing player loses credits. The change of credits will be displayed on a pop-up window when the game ends.

1. **Code Implementation**

* Server-side implementation:
  + Class GameManager: Add a method to calculate credits change.
    - calculate\_credit\_change(self, C\_w, C\_l, S, base\_reward=50): calculates credit changes for winner and loser based on game factors including current credits of the winner and loser, and number of stones placed in the game. The credits changes are calculated as the following:
      * Arguments:
        + C\_w (int): Winner’s current credits
        + C\_l (int): Loser’s current credits
        + S (int): Total stones placed in game
        + base\_reward (int): Base credit reward amount (default: 50)
      * Variables:
        + Credit scale factor: credit\_scale = log((C\_w + C\_l) + 10) / 5
        + Stone efficiency: stone\_efficiency = 1 / sqrt(max(S, 9))
        + Skill factor:

If C\_w >= C\_l, then skill\_factor = 1 / (1 + (C\_w - C\_l) / 200)

If C\_w < C\_l, then skill\_factor = 1 + (C\_l - C\_w) / 200

* + - * Final reward:
        + reward = base\_reward \* credit\_scale \* stone\_efficiency \* skill\_factor
        + Return max(round(reward), 1)
    - \_handle\_make\_move(self, message) & \_handle\_player\_disconnected(self, message): Inside these two functions, the users’ credits are made sure not to be negative after changing.

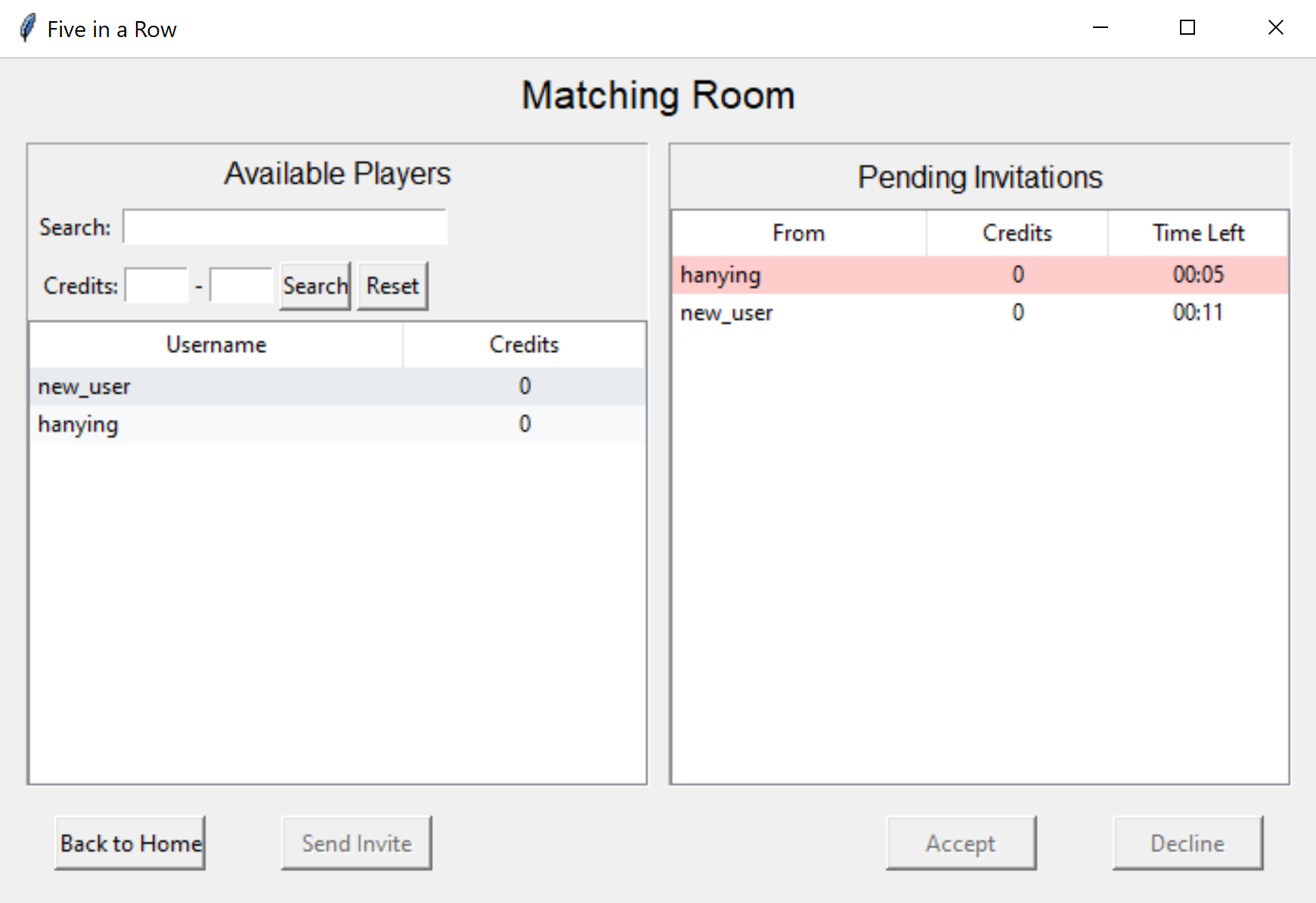
1. **Development Process and Challenges**

* When designing the credits system, we want the magnitude of the credit changes to depend on several in-game factors:
  + Relative Skill Difference: If a lower-credit player defeats a higher-credit opponent, they receive a larger credit bonus as a reward for the upset, while the higher-credit player is penalized more heavily for the loss.
  + Efficiency of Play: If a game ends with fewer stones placed, we think the victory is more decisive, which results in a greater credit adjustment.This promotes aggressive and skillful play.
  + Game Scale: Players with generally higher credit levels will see larger changes in credit adjustments than beginners or lower-ranked players, reflecting the higher stakes of competitive matches at advanced levels.

This logic creates a dynamic and self-balancing system that rewards not only winning but winning skillfully and under challenging circumstances.

* In the beginning, stone\_efficiency was calculated as 1 / sqrt(S). However, this caused an error when S = 0, which can happen if a player exits the game or disconnects at the very start. To address this, we modified the formula to stone\_efficiency = 1 / sqrt(max(S, 9)), which prevents division by zero and avoids large credit changes when a player quits early. We assume the exiting player loses in the minimum possible number of steps, which is 9.

### Matching Room



1. **Functionality Details and User Logic**

The matching room page allows users to view potential opponents currently in the room and manage game invitations in real time. On the left panel, users see a dynamic list of available players, including their usernames and game credits. This list can be refined using a search bar or filtered by credit range, making it easy to find opponents of similar skill or desired competitiveness. This feature enhances the user experience by supporting purposeful matchmaking and accommodating both casual and ranked play. Users can select a player from the list and send a game invitation. Only users who are in the matching room appear in the list of available players, as we assume all users in the room are open to sending or receiving invitations. Each user can send only one invitation at a time and must either cancel the current invitation or wait 15 seconds for it to expire before sending another.

On the right panel, users can view their pending game invitations. Each invitation displays the sender’s username, current credits, and a countdown timer showing how much time remains before the invitation expires. Invitations are valid for 15 seconds, and those with fewer than 5 seconds remaining are highlighted in red to prompt timely responses. Users can select a pending invitation and choose to either accept or decline it using the corresponding buttons below.

* If the user clicks “Accept,” they and the sender will be paired and redirected to a game session.
* If the user clicks “Decline,” the invitation will be removed from the right panel, and the sender will be notified that it was declined.
* If the user does not respond within 15 seconds, the invitation expires and disappears from the right panel, and the sender is notified that it has expired.
* If the user leaves the matching room or accepts an invitation while having other pending invitations, the remaining senders will be notified that the user is no longer available, and their invitations will be automatically declined.

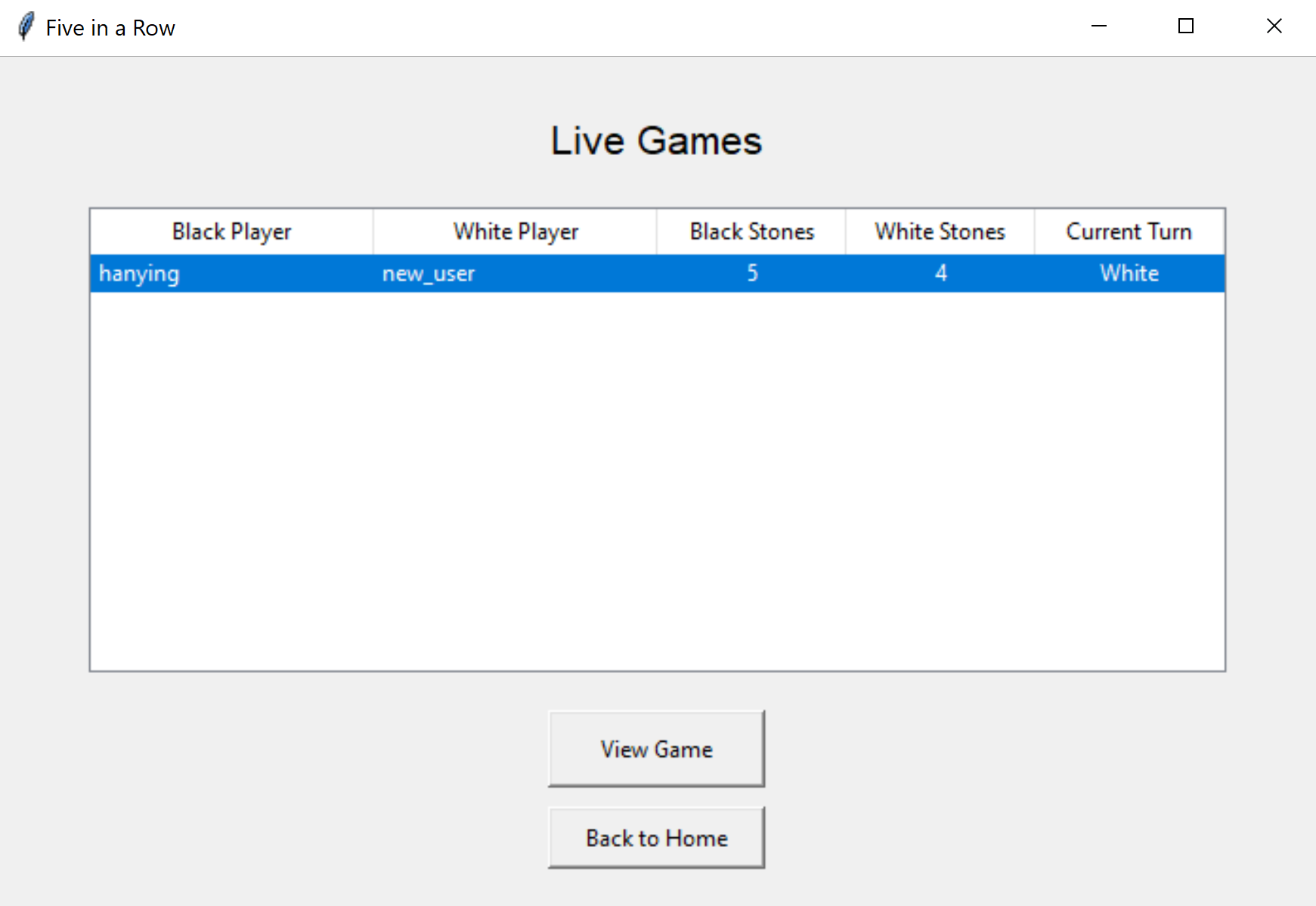
1. **Code Implementation**

* Client-side Implementation
  + Class MatchingRoomUI: Add a separate UI class for the matching room.
    - \_\_init\_\_(self, root, client): initialize matching room UI. Create a left panel for searching and viewing matching room users. Create a right panel for displaying pending invitations. Set up action buttons. Call start\_polling and update\_invitation\_timers.
    - start\_polling(self): send a GET\_MATCHING\_ROOM\_USERS request to the server and schedule the next request after 0.5 seconds.
    - stop\_polling(self): stop polling matching room user list and the invitation timer.
    - search\_users(self): filter the matching room player list by username substring match or credit range. Call update\_users\_list to only show filtered users.
    - reset\_search(self): reset all search filters and display the full player list.
    - return\_to\_home(self): return to the home screen. Stop polling and send a MATCHING\_ROOM\_LEAVE message to the server.
    - update\_invitation\_timers(self): updates the countdown timers for all pending invitations every 0.5 second.Highlights invitations that are about to expire (less than 5 seconds) and removes expired ones.
    - update\_requests\_list(self, requests): updates the invitations list with the latest active match requests sent from the server. Adds new requests and removes expired ones by calling \_remove\_invitation.
    - \_remove\_invitation(self, req\_id): removes a specific invitation from display and tracking.
    - on\_invitation\_select(self, event): handles selection of an invitation from the pending invitations list. Enables the accept/decline buttons when a pending invitation is selected, and disables the buttons otherwise.
    - \_accept\_invitation(self): handles accepting a pending invitation. Sends a MATCH\_RESPONSE message with ‘accepted’=True to server and removes the invitation from display.
    - \_decline\_invitation(self): handles declining a pending invitation. Sends a MATCH\_RESPONSE message with ‘accepted’=False to server and removes the invitation from display.
    - update\_users\_list(self, users): update the player list display with given data. Apply the search filter if it exists. Preserve selection state. If no user is selected, disable the “Send Invite” button.
    - on\_user\_select(self, event): handles selection of a player from the available players list. Enables the invitation button if a play is selected and disables it otherwise.
    - send\_match\_request(self): sends a game invitation to the selected player and shows a waiting page with countdown timer. Sends a MATCH\_REQUEST message to the server.
    - update\_waiting\_timer(self, to\_user): updates the countdown timer in the waiting page every 0.5 second.Pops up a message box to notify the user if the invited player leaves the matching room and cancels the match request automatically. If the invitation has expired, the waiting page will disappear.
    - animate\_dots(self): animate\_dots(self): animation for waiting dots.
    - animate\_spinner(self): animation for loading icon.
    - \_cancel\_waiting(self, user=None, notify=False): cancels a sent pending invitation request by sending a MATCH\_CANCEL message to the server.
    - handle\_match\_response(self, accepted, responder): handles response from the invited player. Redirect to the game session if accepted=True, or notify the user that the invitation is declined if accepted=False.
  + Class GameClient: Add methods to handle matching room related server responses.
    - handle\_server\_message(self, message):
      * Handles MATCHING\_ROOM\_USERS by calling update\_users\_list from MatchingRoomUI.
      * Handles MATCH\_REQUESTS\_RESPONSE by calling update\_requests\_list from MatchingRoomUI class.
      * Handles MATCH\_CANCEL by calling \_remove\_invitation from MatchingRoomUI class.
      * Handles MATCH\_FOUND by calling start\_game and handle\_match\_response from MatchingRoomUI class with accepted=True if the user is in the matching room.
      * Handles MATCH\_DECLINED by calling handle\_match\_response with accepted=False from MatchingRoomUI class.
* Server-side Implementation
  + Class GameServer: Add message handling for client messages sent from matching room UI.
    - \_process\_message(self, message, client\_socket):
      * Handles MATCHING\_ROOM\_JOIN by adding the user to the list of matching room users and returning a MATCHING\_ROOM\_USERS message.
      * Handles MATCHING\_ROOM\_LEAVE by removing the user from matching room users and calling \_cancel\_match\_request to cancel all pending requests associated with the user.
      * Handles GET\_MATCHING\_ROOM\_USERS by creating a dictionary of users in the matching room containing their username and game credits and returning a MATCHING\_ROOM\_USERS message.
      * Handles MATCH\_REQUEST by adding the request information to match\_requests dictionary and returning a MATCH\_REQUESTS\_RESPONSE message to the invited user’s client.
      * Handles MATCH\_REPONSE by returning a MATCH\_FOUND message to the clients of both invitation sender and receiver in order to start a game if accepted=True. Otherwise, calling \_cancel\_match\_request to cancel this invitation and return a MATCH\_DECLINED message to the invitation sender.
      * Handles MATCH\_CANCEL by calling \_cancel\_match\_request to cancel this invitation.
    - \_cancel\_match\_request(self, req\_id, from\_user, to\_user, notify): if req\_id is given, delete the request. If from\_user and to\_user are both given, delete the request that matches the sender and receiver information. If to\_user is given and notify=True, send a MATCH\_CANCEL message to the receiver.
    - \_clean\_expired\_requests(self): removes expired match requests from match\_requests dictionary every second.

1. **Development Process and Challenges**

* One problem we encountered during development was that the filtered results in the user list panel were not persistent. The filtered user list will appear for a second and then the list changes back to the unfiltered one immediately. We solved this problem by applying the search filter dynamically and applying the filters every time we call update\_users\_list. In addition, we made the selection in the list persistent, such that if the user selects a player in the original list, and after applying the filter the selected player is still there, the selection will be automatically populated. These enhancements were implemented to achieve a smoother user experience.
* After developing the feature for retrieving available users in the matching room—along with search and reset options on the left panel—we began implementing the invitation system. Initially, we created a separate polling function for the right panel to update pending invitations. However, we later decided to have the server actively send a MATCH\_REQUESTS\_RESPONSE to the receiver whenever a new invitation arrives. The client then calls the update\_requests\_list method upon receiving this message. By removing the second polling mechanism, we reduced resource usage and latency while also avoiding potential conflicts from having two concurrent polling loops on the same page. We used a recursive function, update\_invitation\_timers, on the client side to update countdown timers for each pending invitation and to remove expired invitations.
* We then encountered a bug where an invitation would reappear with a 15-second countdown after it had expired. This happened because expired invitations were not properly removed on the client side. To fix this issue, we added logic to update\_invitation\_timers to check whether an invitation has expired and, if so, call \_remove\_invitation. We also noticed that expired invitations remained in the server-side match\_requests dictionary. To address this, we implemented a \_clean\_expired\_requests function, which is called by the \_check\_timers helper method that updates server state every second.
* Another issue occurred when users A and B sent invitations to user C at roughly the same time. If C accepted A’s invitation or left the matching room, B would remain stuck on the waiting page. Moreover, users who had already entered a game still appeared in the available player list. To fix this, we removed users who either entered a game or left the matching room (including disconnections) from the server-side matching\_room\_users set. On the client side, when updating the countdown timer during the waiting state, we added logic to check if the target user is still in the matching room. If not, the sender is notified that the receiver has left, and the invitation is automatically canceled. This prevents the sender from wasting 15 seconds waiting for someone who can no longer respond.
* Initially, we sent a MATCH\_CANCEL message to clients when the server received a response from the invitation receiver. This was aimed to clear out the match request after it had been handled, but it caused confusion and unwanted pop-ups for users who had already started a game. To improve the user experience, we introduced a new handle\_match\_response function on the client side. If an invitation is accepted, the server sends a MATCH\_FOUND message to both users, and handle\_match\_response updates the UI of the waiting page. If the invitation is declined, a new message type MATCH\_DECLINED is sent instead, and handle\_match\_response notifies the sender appropriately. Now, MATCH\_CANCEL is only sent when the sender explicitly cancels the request, and only the receiver’s client handles its removal from both frontend and backend.
* While testing edge cases, we discovered that if the invitation sender disconnected from the server immediately after sending an invitation, the request still appeared on the receiver's panel. This caused an error if the receiver tried to accept it. To handle this, we added logic to detect and cancel any active outgoing invitations when a user disconnects. The server calls \_cancel\_match\_request, and a MATCH\_CANCEL message is sent to the receiver to remove the now-invalid invitation.
* Another small issue we noticed while developing the function was that there was a roughly 1-second discrepancy between the countdown timers shown to the sender and receiver. Upon review, we found this was due to a mismatch in how expiration time was calculated: the sender used the time the invitation was sent plus 15 seconds, while the receiver used the time the message was received plus 15 seconds. To resolve this, we included the exact expiration time in the MATCH\_REQUEST message so that both sender and receiver use the same reference. Although a small discrepancy (<0.5s) still exists—likely due to minor differences in timer update intervals between clients—we consider this acceptable since the expiration time is now consistent across both sides.

### Live Game Viewer



1. **Functionality Details and User Logic**

The Live Game Viewer function allows users to watch ongoing games in real time. When users click the “View Live Games” button in the homepage, they are first presented with a list of currently active games, each showing player names, the total number of stones placed so far, and the current turn. This list updates automatically as new games start or existing games finish. Users can select any game from the list and click the “View Game” button to open a live board view. The game board then displays the current state of the selected game and updates automatically as players make moves. Users can only view one game at a time, and they have to close the current game board in order to select and view another live game. This feature enhances engagement by allowing spectatorship and makes it easy for users to follow competitive matches or learn strategies from other players in real time.

****

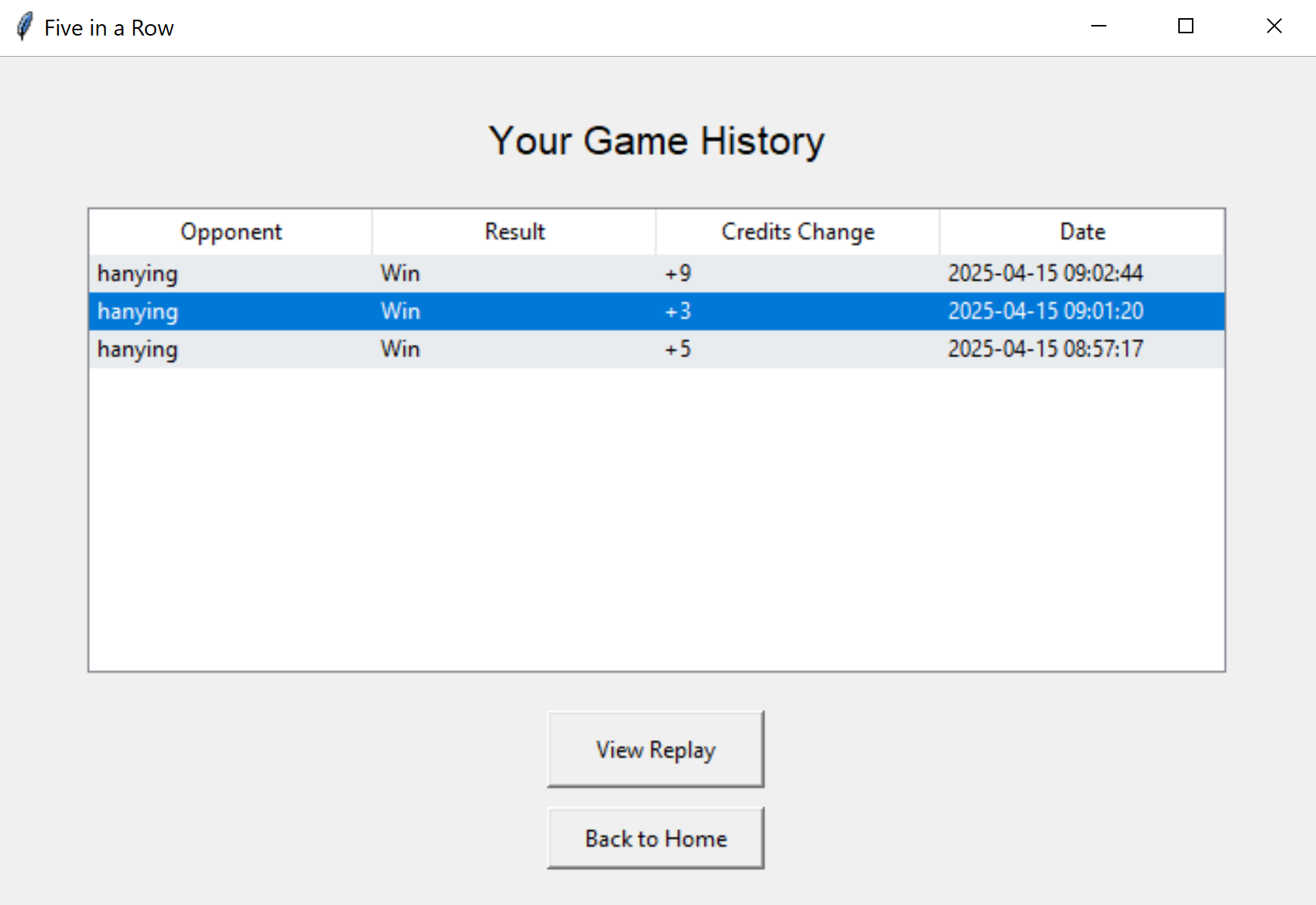
1. **Code Implementation**

* Client-side Implementation
  + Class LiveGamesUI: Add a separate UI class to view the list of current live games.
    - \_\_init\_\_(self, root, client): Initializes the live games interface. Display a list of current ongoing games.
    - poll\_live\_games(self): requests updates of live games from server every second by sending a GET\_LIVE\_GAMES\_REQUEST message. Manages game selection state and polling interval.
    - stop\_polling(self): stops the live games polling timer.
    - return\_to\_home(self): handles returning to homepage. Ensures that the active viewer has closed before returning.
    - load\_live\_games(self, games): updates the UI with the latest list of live games.
    - on\_row\_click(self, event): handles selection of a game from the list. Enables the “View Game” button only when a valid game is selected.
    - is\_game\_ended(self, game\_id): checks if a game has ended.
    - view\_game(self): opens a live viewer for the selected game. Ensures only one viewer is open at a time.
  + Class LiveGameViewerUI: Add a separate UI class to view the selected game. Provides real-time updates of the game state and handles game completion scenarios.
    - \_\_init\_\_(self, root, client, game\_id, initial\_data, parent\_ui): initializes the live game viewer interface. Created a GameUI with view\_mode=True to show the board and all the updates.
    - poll\_game\_state(self): requests game state updates from the server every 0.25 second by sending a GET\_GAME\_STATE message to the server. Call handle\_disconnect function if no updates are received for more than 2 seconds.
    - update\_game\_state(self, state): updates the game display with new state from server. Calls \_handle\_final\_state(state) to handle game over if the game has ended but the viewer page has not handled the case. If the game hasn’t ended, call game\_ui.update\_game\_state(full\_state, view=True) in the GameUI to update the game board of the viewer.
    - \_handle\_final\_state(self, state): processes the final game state when the game ends. Calls \_show\_game\_over\_popup(winner) to display the winner.
    - handle\_disconnect(self): handles player disconnection scenario by popping up a notification window.
    - \_show\_game\_over\_popup(self, winner): shows the game over popup with winner information.
    - close\_viewer(self): cleans up when closing the viewer. Stops polling and notifies parent UI.
  + Class GameClient: Add method to handle live game viewer server response.
    - handle\_server\_message(self, message):
      * Handles GET\_LIVE\_GAMES\_RESPONSE message by saving the latest live\_games list and calling load\_live\_games function from LiveGamesUI class to update LiveGamesUI.
      * Handles GAME\_STATE message if both the live\_game\_ui and live\_game\_viewer\_ui exist. Calls the update\_game\_state function from the LiveGameViewerUI class to update the board.
* Server-side Implementation:
  + Class GameServer: Add message handling for client messages sent from live game UI and live game viewer UI.
    - \_process\_message(self, message, client\_socket):
      * Handles GET\_LIVE\_GAMES\_REQUEST by calling handle\_message from Class GameManager.
      * Handles GET\_GAME\_STATE by calling handle\_message from Class GameManager.
  + Class GameManager: Add methods to handle GET\_LIVE\_GAMES\_REQUEST and GET\_GAME\_STATE.
    - \_\_init\_\_(self, user\_manager): initialize the game manager with a user manager instance.
    - handle\_message(self, message):
      * Handles GET\_LIVE\_GAMES\_REQUEST by calling \_handle\_get\_live.
      * Handles GET\_GAME\_STATE by loading live games from the live games database by calling load\_live\_game function in Database class. Return a GAME\_STATE message.
    - \_handle\_get\_live(self): retrieves information about currently active games by calling get\_live\_games function in Database class and returning a GET\_LIVE\_GAMES\_RESPONSE message.
  + Class Database: Add methods to manage live games data.
    - get\_live\_games(self): retrieves summary information about all currently active games by calling \_load\_live\_games.
    - load\_live\_game(self, game\_id): loads the current state of an active game from the database by calling \_load\_live\_games.
    - \_load\_live\_games(self): internal method to load all active game states from the live games file and handles file corruption by calling \_get\_prev\_live\_info function if needed.
    - \_get\_prev\_live\_info(self): retrieve the previous live game information.

1. **Development Process and Challenges**

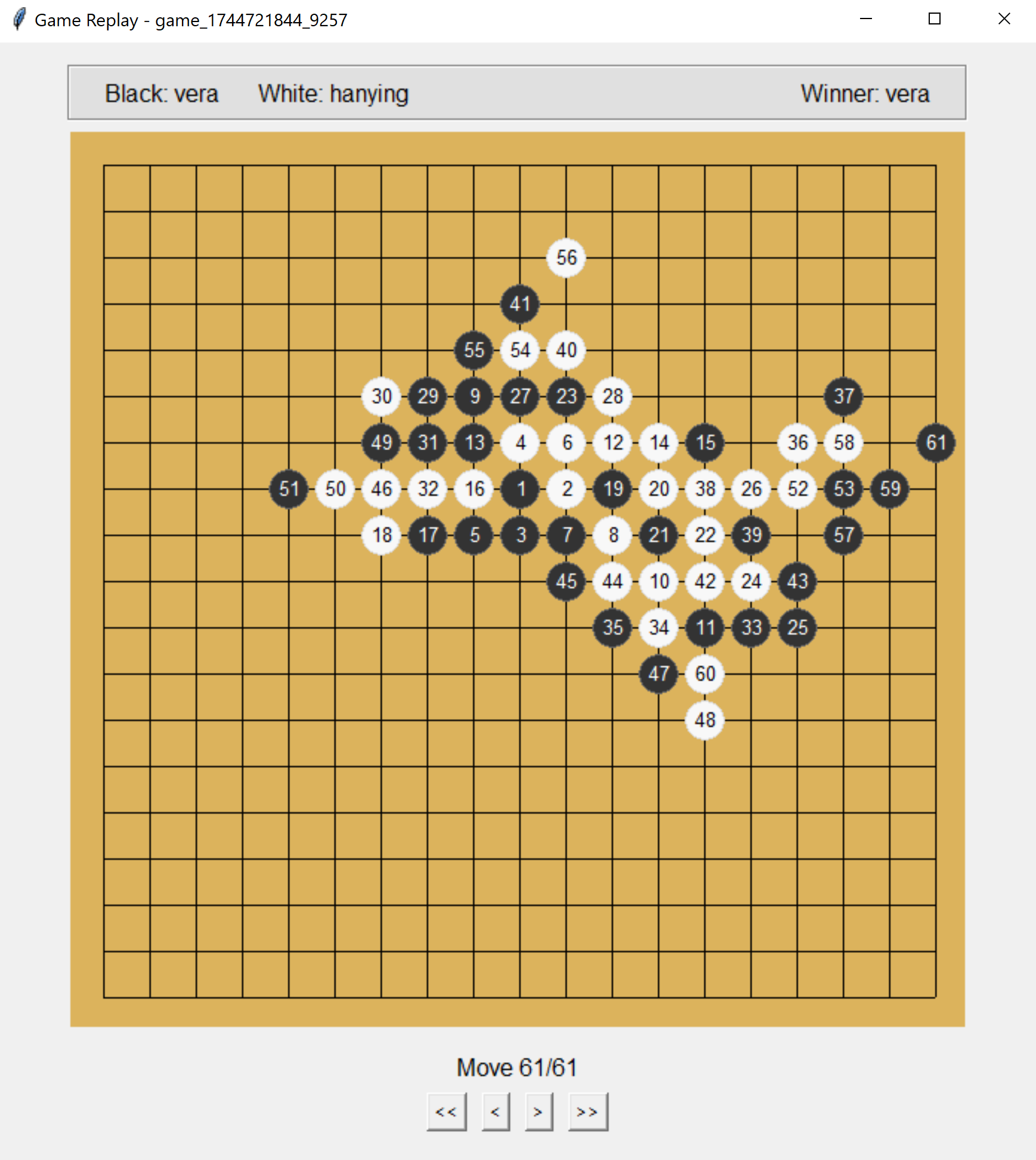
* We first implemented the LiveGamesUI class to display all ongoing games with basic information such as usernames and the number of stones placed so far. We added a polling function to retrieve data every second and created the get\_live\_game function in the Database class on the server side, which reads and summarizes live game information directly from the live games file. However, when clicking on a game listed on the page, the selected game could only be highlighted for one second, requiring users to click again to re-highlight it. We discovered this issue was caused by the polling function, which cleared the highlight every time the page was updated. To fix this, we checked whether a game was selected and ensured it was re-highlighted every time the load\_live\_games function (called during polling) was executed. This provides a consistent UI interface.
* After selecting a live game and clicking the “View Game” button, a board pops up showing the current state of the selected game. To avoid redundant code, we reused the GameUI class with view\_mode=True to draw the board and stones in real-time. The biggest challenge in this feature was updating the board state properly. Initially, only the board appeared—no stones were shown, and it never updated. To fix this, we added a polling function poll\_game\_stat in the LiveGameViewer class, which sends a GET\_GAME\_STATE message every 0.25 seconds to retrieve the latest game data. When the client receives the response, it calls update\_game\_state in the LiveGameViewer class to organize the data and then calls game\_ui.update\_game\_state to reflect the updates on the live game viewer interface. In the GameUI class, when viewer\_mode=True, we ensured that users cannot place stones on the board since they are not players in the game, and we modified parts of the UI to improve the viewer experience. With these functions in place, the board updates correctly, and users can smoothly view live games.
* We also encountered a bug where, after the game ended, the viewer’s client sometimes failed to update the last stone placed and did not show the game-over window. We realized this was because the client stopped polling immediately after receiving ‘game\_over’=True in the game state data, but the LiveGameViewerUI page may not have updated its internal status in time and therefore failed to call the \_show\_game\_over\_popup function. To resolve this, we added the \_handle\_final\_state function to handle the case where the received game state data contains ‘game\_over’=True but the internal self.game\_over is still False. This function updates self.game\_over to True, adds the final stone to the board, and calls \_show\_game\_over\_popup to display the game result.
* In the scenario where one player disconnects, the game ends for both players. However, the viewer’s board would freeze without any notification. To address this, we added a variable self.last\_update\_time to record the time when polling data was last received. If no data is received for more than 2 seconds, we call the handle\_disconnect function to display a popup notifying the viewer that one player has disconnected and the game has ended. This improves the viewer’s experience by providing timely feedback about game termination.
* Since we store game data in a JSON file and retrieve live game information by reading this file, which might be written to by the server at the same time, this sometimes causes a JSON file corruption error. Previously, we returned an empty dictionary {} when this error occurred in the \_load\_live\_games function in the Database class on the server side. However, this caused inconsistency in the LiveGamesUI, where users would briefly see a live game record disappear for one second (when the error occurred) and then reappear in the next polling cycle. To fix this issue, we introduced two variables: self.\_prev\_live\_info and self.\_prev\_live\_info\_lock in the Database class to cache the most recent live game information that was read successfully. If an error occurs during reading, the cached data is returned instead. This prevents sudden disappearances of live game records in the UI and ensures a smoother user experience.

### Game History



1. **Functionality Details and User Logic**

The Game History and Replay feature allows users to review their completed matches and analyze past gameplay. After clicking the “Game History” button in the homepage, the users will be directed to the history UI page that displays a list of all previous games in reverse chronological order. Each entry shows key game information such as opponent name, game result (win or loss), credit changes, and the date the game was played. When a user selects a specific game from the list, the “View Replay” button becomes active, and an empty game board is initialized for playback after clicking it. Users can step through the match move-by-move using forward (“>”) and backward (“<”) buttons or watch an automatic playback using the fast-forward (“>>”) and rewind (“<<”) controls. Each stone is labeled with the number of steps to indicate their order of placement, helping users visualize the flow of the match. This feature supports both casual review and strategic analysis, making it easy for players to reflect on their performance and learn from past games.



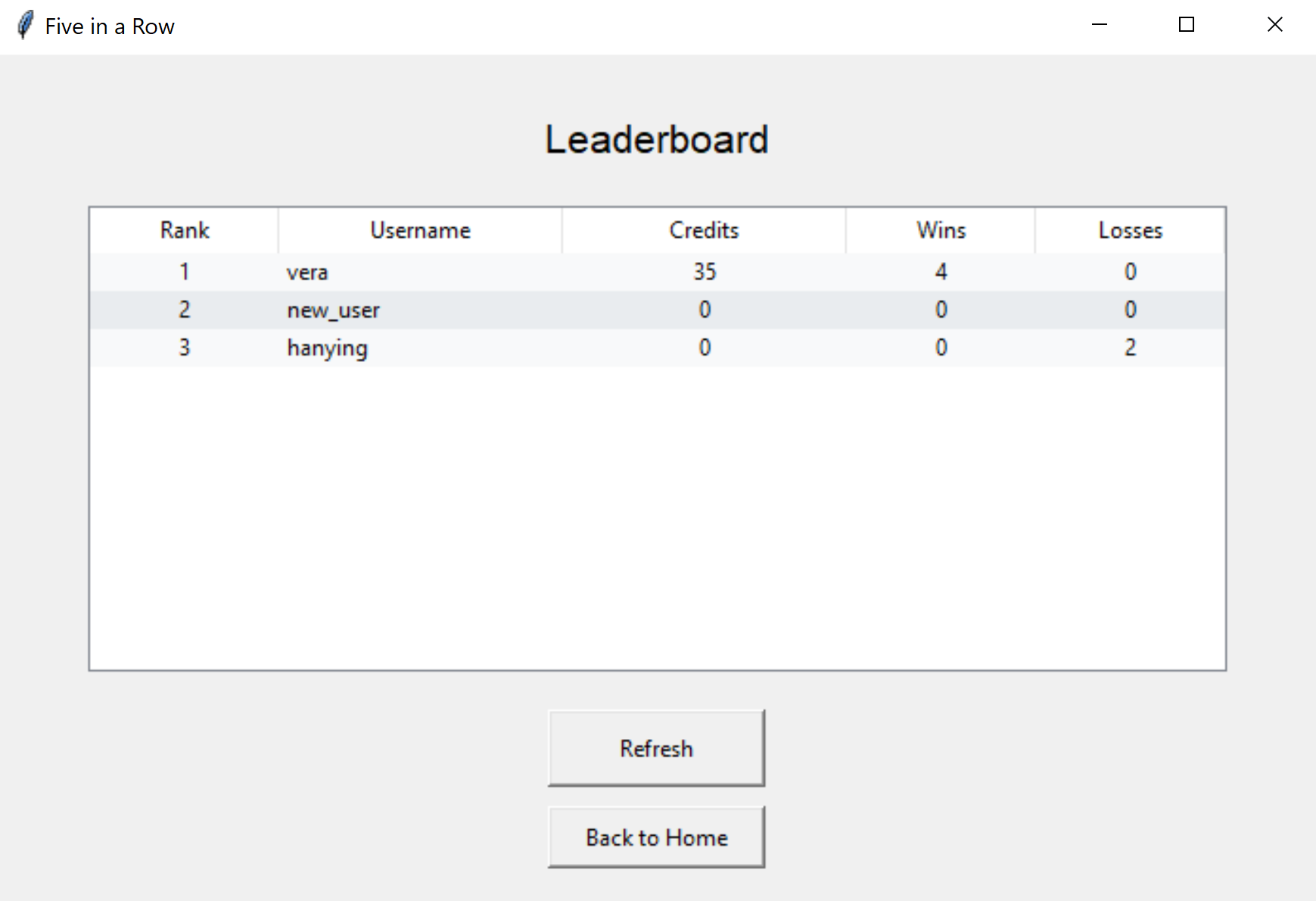
1. **Code Implementation**

* Client-side Implementation
  + Class HomeUI: The main home screen UI.
    - show\_history(self): opens the game history interface and sends a GET\_HISTORY\_REQUEST message to the server.
  + Class HistoryUI: Add a separate UI class to view the list of past games in reverse chronological order.
    - \_\_init\_\_(self, root, client): initializes the game history interface. Display a list of past games in reverse chronological order.
    - load\_history\_from\_server(self, histories): update the latest game history list.
    - on\_row\_click(self, event): handles selection of a game from the history list. Enables the replay button when a valid game is selected.
    - show\_replay(self): launches a GameReplayUI if there is a selected game and provides the selected game data.
  + Class GameReplayUI(GameUI): Add a separate UI class to view the replay of the selected game. Inherits from GameUI class to reuse board drawing and stone placement logic.
    - \_\_init\_\_(self, root, client, game\_data): initializes the game replay interface.
    - create\_replay\_controls(self): creates and arranges the replay control buttons and status display.
    - start\_replay(self): starts the automatic replay of the entire game from the beginning (the “>>” button) by calling reset\_board and replay\_whole\_game functions to clear the board and start replay.
    - replay\_whole\_game(self): automatically plays through all moves with a 0.5 second delay between moves by recursively calling next\_move function until all the stones have been placed. Disable all the control buttons except for the reset button (“<<”).
    - reset\_board(self): resets the board to its initial empty state and stops any ongoing replay. Update the board by calling update\_display.
    - previous\_move(self): steps backward one move in the replay sequence. Update the board by calling update\_display.
    - next\_move(self): steps forward one move in the replay sequence. Update the board by calling update\_display.
    - update\_display(self): updates the board display and moves counter text. Calls parent class’s (GameUI) update\_game\_state function to update the board and place stones.
    - toggle\_buttons(self, enable=True): enables or disables the control buttons during replay.
    - safe\_close(self): safely handles window closing by stopping any ongoing replay first.
  + Class GameClient: Add method to handle game history server response.
    - handle\_server\_message(self, message): handles GET\_HISTORY\_RESPONSE message by saving the latest game history list data and calling load\_history\_from\_server function in HistoryUI class.
* Server-side Implementation:
  + Class GameServer: Add message handling for client messages related to game history.
    - \_process\_message(self, message, client\_socket): handles GET\_HISTORY\_REQUEST message by calling handle\_message function in GameManager class.
  + Class GameManager: Add method to handle GET\_HISTORY\_REQUEST.
    - handle\_message(self, message): handles GET\_HISTORY\_REQUEST by calling \_handle\_get\_history.
    - \_handle\_get\_history(self, message): retrieves game history for a specific player by calling get\_user\_history function in Database class. Return a GET\_HISTORY\_RESPONSE message to the client.
  + Class Database: Add methods to retrieve game history data from the server database.
    - get\_user\_history(self, username): retrieves all game histories for a specific user by calling a help function \_load\_game\_histories.
    - \_load\_game\_histories(self): internal method to load all game histories from the game file.

1. **Development Process and Challenges**

* We first implemented the HistoryUI class to display all game history for the user, including basic information such as the opponent’s username, win/loss result, credit changes, and timestamp. When a user selects a game and clicks the “View Replay” button, a GameReplayUI class is created and an empty board pops up. To avoid redundant code, the GameReplayUI class inherits from the GameUI class and sets replay\_mode=True. Initially, an “Exit Game” button was shown at the bottom of the replay UI page because it was inherited from GameUI. However, we realized that clicking this button would result in an error, as it sends a PLAYER\_DISCONNECTED message to the server. To fix this, we modified the code in the GameUI class to ensure the “Exit Game” button is not displayed when replay\_mode=True.
* We initially used a list (move\_history) to store move history data on the server side. However, we later discovered that the saved move history could become mixed up if two games were played simultaneously. To address this, we changed move\_history to a dictionary, where the key is the game\_id and the value is the corresponding list of move history data. This change ensures that game data is saved correctly even when multiple games are happening concurrently.
* We also made several small modifications to the GameUI code when replay\_mode=True to provide a better UI experience during game replays. If replay\_mode is enabled, the color assignment window will not appear, and the winner information will be displayed at the top of the page in the same location originally used for the countdown timer. These UI adjustments help improve the overall replay experience for users.

### Leaderboard



1. **Functionality Details and User Logic**

The leaderboard function provides a ranked overview of player performance across the game platform. It displays a list of top 100 players sorted primarily by their total game credits, which reflects both skill and consistency. For each player, the leaderboard shows their rank, username, current game credits, number of wins, and number of losses. This allows users to quickly compare their standing against others and see who the top players are. When users click the “Refresh” button, the latest leaderboard will be retrieved and displayed. Users can scroll through the list to view rankings or locate specific players, encouraging motivation and engagement within the game’s competitive ecosystem.

1. **Code Implementation**

* Client-side Implementation
  + Class HomeUI: The main home screen UI.
    - show\_leaderboard(self): opens the leaderboard interface showing player rankings and sends GET\_LEADERBOARD\_REQUEST message to the server.
  + Class LeaderboardUI: Add a separate UI class to display top 100 players rankings based on credits and wins.
    - \_\_init\_\_(self, root, client): initializes the leaderboard interface. Display a list of top 100 players sorted primarily by their total game credits.
    - load\_leaderboard(self, leaderboard\_data): loads and displays the latest leaderboard data.
    - refresh(self): refresh the leaderboard with the latest data by sending a GET\_LEADERBOARD\_REQUEST message.
  + Class GameClient: Add method to handle leaderboard server response.
    - handle\_server\_message(self, message): handles GET\_LEADERBOARD\_RESPONSE message by saving the latest leaderboard data and calling load\_leaderboard function in LeaderboardUI class.
* Server-side Implementation:
  + Class GameServer: Add message handling for client messages related to leaderboard.
    - \_process\_message(self, message, client\_socket): handles GET\_LEADERBOARD\_REQUEST message by calling get\_leaderboard function in UserManager class and returns a GET\_LEADERBOARD\_RESPONSE message.
  + Class UserManager: Add methods to handle GET\_LEADERBOARD\_REQUEST.
    - get\_leaderboard(self, limit=100): generates a ranked leaderboard (list of dictionaries) of players.

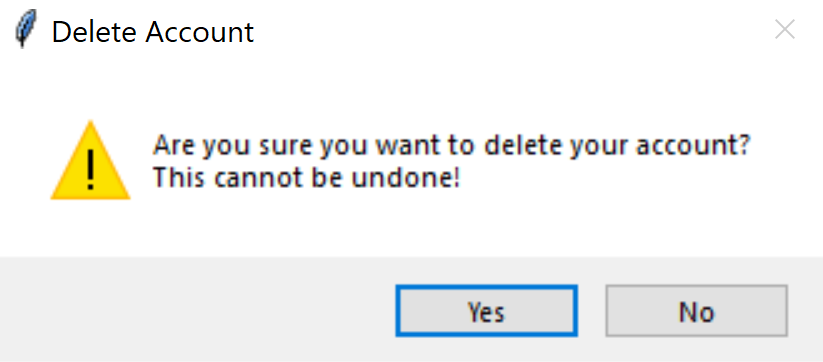
1. **Development Process and Challenges**

* The leaderboard function is relatively simple compared to the other functionalities. The logic is to retrieve all user information from the server database and sort the users by game credits (in descending order), number of wins (descending), and number of losses (ascending). Initially, the leaderboard page only had a “Back to Home” button. Later, we realized that since there was no polling function on the page, the only way users could view the updated leaderboard was by returning to the homepage and then navigating back to the leaderboard page, which was inconvenient. To improve this, we added a “Refresh” button that allows users to manually update the leaderboard and view the latest rankings.

### Delete Account

1. **Functionality Details and User Logic**

On clicking “Delete Account” on the homepage, a pop up window will appear to ask the user to confirm account deletion. If “no”, no changes will be made. If “yes”, the user’s account is deleted and all associated data is removed. The user will be brought back to the authentication UI. The user’s username will show as “account deleted” in all game histories when viewed by other users.



1. **Code Implementation**

* Client-side Implementation
  + Class HomeUI:
    - delete\_account(self): pop up a message box to ask the user to confirm deletion. If “yes”, stop polling homepage stats and send an ACCOUNT\_DELETE\_REQUEST message to the server.
  + Class GameClient: The game client class for handling responses from the server. Add methods to handle ACCOUNT\_DELETE\_RESPONSE.
    - handle\_server\_message(self, message):
      * Handles ACCOUNT\_DELETE\_RESPONSE by showing home UI, showing a message box indicating success, and current user cleanups.
* Server-side Implementation
  + Class GameServer:
    - \_process\_message(self, message, client\_socket):
      * Handle ACCOUNT\_DELETE\_REQUEST by calling handle\_message from class UserManager.
  + Class UserManager: Add method to handle ACCOUNT\_DELETE\_REQUEST
    - handle\_message(self, message):
      * Handles ACCOUNT\_DELETE\_REQUEST by calling \_handle\_delete\_account, which changes the user’s username to “account deleted” in all game histories, remove user from the user database, and returns an ACCOUNT\_DELETE\_RESPONSE message.

1. **Development Process and Challenges**

* One problem we encountered when developing account deletion is that there would be pop-up windows saying “lost connection to server” when returning to the authentication UI. We found that this error occurred because after the account is deleted and the current username is set to None, the polling process from the matching room UI is not stopped and still polls for stats that contain the username entry, even though the user has already been removed. As a result, the lost connection error occurs every time the user tries to delete their account after having entered the matching room. Thus, we solved this problem and enhanced the implementation by ensuring a smooth exit from the matching room and stopping polling before exiting the matching room UI.

### Log Out

1. **Functionality Details and User Logic**

On clicking “Log Out” on the homepage, the user will be logged out of their current account and will be redirected to the authentication page. In addition, if a user disconnects from the server, the user is logged out and leaves the queue automatically.

1. **Code Implementation**

* Client-side Implementation
  + Class HomeUI:
    - logout(self): send a LOGOUT message to the server, clear the current client state, and return to the authentication UI.
* Server-side Implementation
  + Class GameServer:
    - \_process\_message(self, message, client\_socket):
      * Handle LOGOUT by calling handle\_message from class UserManager.
    - \_handle\_client(self, client\_socket): if a user disconnects from the server, call UserManager to handle a LOGOUT message to log out the user. In addition, call MatchMaking to handle a QUEUE\_REQUEST message with action=leave, to remove the user from the matching queue.
  + Class UserManager: Add method to handle LOGOUT
    - handle\_message(self, message):
      * Handles LOGOUT by calling \_handle\_logout, which discards the logged in user and returns a LOGOUT message.

1. **Development Process and Challenges**

* One bug we encountered was that the number of online players did not change as a user logs out. We found that the user needs to be removed from the current logged in users to reflect the change in the number of online players.

## Test

We conducted a comprehensive unittest for the game application, including tests on shared models, database, user manager, match making, game manager, game server, client UIs, and game client. The tests cover all classes and functionalities provided by the application. We also tried to come up with different edge cases to test the correctness of error handling. Below is a list of our test classes and a description of the tests conducted by each.

* Class TestSharedModels:
  + Test user model: test if user model attributes are correctly set and data dictionary has the correct format.
  + Test game history model: test if game history attributes are correctly set and history dictionaries have the correct format. Verifies the list of moves is properly stored.
  + Test game state model: test game board initialization, players turn checking, and serialization
  + Test message protocol: test message types and message data. Test round trip serialization
* Class TestDataBase:
  + Test user operations: test user saving and user data retrieval. Test nonexist users.
  + Test game history operations: test game history saving and retrieval. Test user-specific game history filtering.
  + Test live game operations: test live game state saving and loading. Test game state deletion.
  + Test get live games: test retrieval of all live games.
* Class TestUserManager:
  + Test signup and login: test user registration, duplicate username prevention, invalid usernames, login authentication, valid and invalid password, proper logout.
  + Test get stats: test user statistics retrieval with correct information and message format.
  + Test account deletion: test successful account removal and cleanups.
  + Test leaderboard: test leaderboard generation and ordering.
* Class TestMatchMaking:
  + Test queue operations: test joining and leaving queue, queue size tracking, and proper response messages.
  + Test match found: test match found notifications, queue cleanup, and single player waiting.
* Class TestGameManager:
  + Test game creation: test game initialization, stone assignment, and initial game state.
  + Test valid move: test board changes and turn switches.
  + Test invalid move: test invalid move rejection.
  + Test win condition: test win detection and game over handling.
  + Test disconnect handling: test game over handling when player disconnects.
  + Test history retrieval: test retrieving user-specific history.
* Class TestGameServer:
  + Test connect to server: test server connection.
  + Test signup and login: test signup and authentication. Test queue requests from logged in users.
  + Test match making: test joining queue and player pair-up.
* Class TestClientUI:
  + Test auth UI: test the authentication UI message sending and error handling.
  + Test home UI: test home screen functionalities and polling.
  + Test waiting UI: test waiting screen functionalities and leaving queue.
  + Test game UI: test valid and invalid moves, turn validation, click handling.
  + Test history UI: test history data loading.
  + Test game replay UI: test move navigation, replay controls, board updates.
  + Test live games UI: test live games data loading.
  + Test live games viewer UI: test viewer initialization and live game state.
  + Test leaderboard UI: test leaderboard data loading.
  + Test matching room UI: test available user list and appropriate invitation handling.
* Class TestGameClient:
  + Test client connection: test proper client-server connection.
  + Test send message: test correct message sending via sockets.
  + Test handle server message: test correct handling of server messages.
  + Test UI transitions: test proper transitions between UI states.

## Documentation

Finally, we added comments and docstrings throughout the codebase to improve readability and maintainability. Each class, method, and function includes a clear description of its purpose, inputs, and outputs to help future developers understand and navigate the code easily. In addition, we created a comprehensive README.md file that explains how to set up and run the system and outlines the main features. We also wrote a detailed engineering notebook that records the entire development process, including design decisions, encountered bugs, technical challenges, and how they were resolved. This documentation serves as a valuable reference for future improvements, debugging, or extension of the system.

## Furthermore

Due to the limited time available for the final project, there are several issues and improvements that could be addressed in future work:

* **Add Replication**: Replication is important in a live match-based game—if the server shuts down, all ongoing games would be forcibly terminated and game data would be lost, leading to a poor user experience. However, adding replication to our Five in a Row game is challenging, as the servers would need to synchronize not only static user and game history data, but also real-time live game state. Additionally, maintaining a consistent UI experience during a server failover is another complex problem. After implementing the core functionalities proposed in our initial design (such as starting a game and reviewing game history), we chose to prioritize developing additional features to enrich the gameplay and explore distributed systems concepts, rather than implementing replication. These features include the matching room, live game viewer, leaderboard, and account deletion. We still plan to explore the possibility of adding replication in the future when more time is available.
* **Synchronize Timers**: As mentioned earlier in the section on the matching room, although the invitation expiration time is synchronized for both the sender and receiver, a small discrepancy (less than 0.5 seconds) still exists in their countdown displays. This occurs because the countdown is updated on the client side, and the update intervals may not align perfectly between clients. We think reducing the interval between updates (currently every 0.5 seconds) could help, but this would also consume more resources. Another approach could be to have the server send the countdown value directly to both the sender and receiver, which would improve synchronization but require more frequent communications between the clients and server. Since both the sender and receiver share the same expiration time, we believe this is a minor issue, but further exploration may help improve precision and user experience.
* **Use a More Advanced Database**: We currently use JSON files to store user data, past games, and live game state. However, during development, we encountered JSON file corruption issues, especially with the live game file which is updated frequently during gameplay. These read/write conflicts led to inconsistencies in the live game viewer UI. As noted in the Live Game Viewer section, we introduced two variables to cache the last successfully read data and ensure UI consistency, but this workaround can cause delays in updates. As the number of users increases, this issue may worsen. A better long-term solution would be to migrate to a more advanced database system, such as SQLite, which supports concurrent access and can eliminate this issue at the root. We plan to explore alternative database solutions in the future.
* **Use a More Efficient Communication Protocol**: Our current system uses a JSON wire protocol for communication between the server and clients. Although this protocol is simple and easy to implement, it is not very efficient. We initially also tried gRPC when developing the system, but we found that certain functionalities were more complex to implement and encountered bugs we were unfamiliar with. As a result, we decided to continue using the JSON protocol to focus on feature development. In the future, we hope to revisit and explore more efficient communication protocols such as gRPC or other approaches to improve performance and reduce resource consumption.
* **Deploy as an Online Website or Downloadable Software**: Currently, the system runs only on devices with Python installed, and both the server and clients must be on the same network due to IP restrictions. In the future, we would like to deploy the program as an online website or standalone software, so users can access the game without needing Python. Additionally, we hope to remove the limitation that requires the server and clients to be on the same local network, making the system accessible over the internet.

These potential improvements highlight the scalability and extensibility of the system. While we successfully implemented a wide range of core features within a limited timeframe, there are still many opportunities to enhance the performance, robustness, and user experience of the platform. This engineering notebook documents not only what we built, but also lays the groundwork for continued development in the future.