PLANNING

Online Multiplayer Board Game Platform

W2025

Team Establishment

February 24, 2025

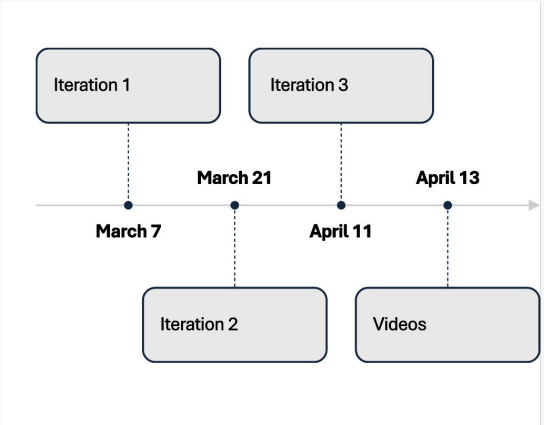
Team	Size
Game Logic	5
GUI	4
Networking	4
Authentication & Profile	4
Leaderboard & Matchmaking	4
Integration	4

In order to best approach each objective, our group is split up into 6 sub-teams. Each sub-team size, is chosen accordingly to the demands of the task at hand. Dividing and conquering is a formidable approach, as it allows to have all hands on deck, ensuring no area remains neglected.

To best complete Iteration 1, each team is assigned specific tasks. Each task is assigned based on workload, and relativity to Iteration deliverables.

Iteration 1 Key Deliverables

- Planning Docs
- Gitlab_Link.txt
- class_diagram.png/svg
- use_case_descriptions.pdf



Iteration 1 Group Tasks

As mentioned before, each team was assigned specific tasks for completion. The idea behind this, was that once each team was complete with their work, work could be evaluated and brought together. This allows us to eliminate any duplicates at the end, and get a visual on the most important components of the project.

GAME LOGIC TEAM

Games to be implemented:

- Tic Tac Toe
- Connect Four
- Checkers

The Game Logic Team is responsible for establishing the use cases for the stated games above. Their job is to create thorough descriptions, as well as diagrams, to model the games on our platform. This role is vital, as functionality on other teams has dependencies on the way these games work. The Game Logic Team is responsible for completing these use cases, and class diagrams, all by March 2, 2025, to ensure each corresponding team can include relevant logic towards their cases and diagrams.

GUI TEAM

The GUI Team is responsible for developing the visuals for the platform. Their use cases contain: *Welcome page, Game Dashboard, Selecting game from the available library, Moving pieces, Joining game, View profiles, Challenge profiles, In-game chat, Display leaderboard, Quit option, Menu Option, Log in, Create Account, Winning page, Losing page, Tie page, Log out, Settings Option.* These are all vital, and fundamental components for the platform. These use cases, use case diagrams, and overall class structure diagram should also be complete by March 2, to give them sufficient time to start working on GUI mock-ups.

Iteration 1 Group Tasks

NETWORKING TEAM

The Networking Team is responsible for establishing the features that allow users to use online features, and communicate with the online components of each game. Not only do they allow each player to connect to the game, but to other players and friends. They are responsible for devising the follow use cases: *Join Match, Log Game Result, Update Leaderboard Rank, Join Match as a Party, Immersion, Get Games Catalogue, Verify Server Online Status, Reconnect to Game, Send Player Action (in-game, chat), Disconnect Player.* hese use cases, use case diagrams, and overall class structure diagram should also be complete by March 2, to give them sufficient time to start working on GUI mock-ups.

AUTHENTICATION & PROFILE TEAM

The Authentication & Profile Team are responsible for account management. Their features are integral for access control. They are responsible for the following use cases: *Settings, Login, Change Username, View Your Stats, Change Password, Change Email, Forgot Password or Username, Logout.* These use cases, use case diagrams, and overall class structure diagram should also be complete by March 2, to give them sufficient time to start working on planning, as well as to have the liberty to communicate with other subsections to expand on the scope of functionality. They continuously work with other groups throughout the whole 2 weeks, and ultimately create the final diagrams.

Iteration 1 Group Tasks

LEADERBOARD AND MATCHMAKING TEAM

The Leaderboard and Matchmaking Team are responsible for providing a structured ranking system, and an optimized algorithm matchmaking system based on player rankings. They are responsible for the following use cases: *Player joins matchmaking queue, MMR changes after a game completes, Display leaderboard, Filter the leaderboard on specific values, Updating player stats after each game, Player leaves matchmaking queue, Game ends in tie, Player gets promoted a rank, Player gets demoted a rank, Player leaves game mid match, View match history, Handling MMR tie on leaderboard, MMR decay.* These are all vital, and fundamental components for the platform. These use cases, use case diagrams, and overall class structure diagram should also be complete by March 2, to give them sufficient time to start working on GUI mock ups.

INTEGRATION TEAM

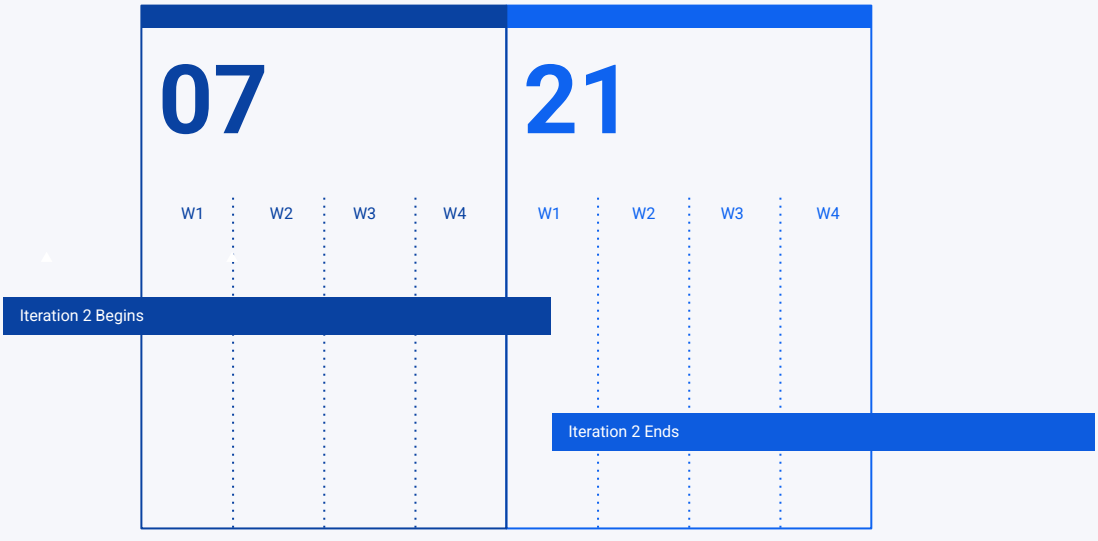
Finally, the Integration Team's job is to tie everything together. They are responsible for combining the use case diagrams, and structure diagrams that each team made. They essentially take all of the sub-implementations and devise a plan to incorporate them all together. Essentially they have a very important role, in tying each thing together. They constantly communicate with other teams, and help where the help is needed. This work should be complete by March 6, a few days after the original diagrams are made to provide them with sufficient time to integrate them all in one system. This way, we would have a full 24-hours to submit with no issues.

Iteration 2 Group Tasks

MOVING FORWARD

Iteration 1 is crucial in providing us with a solid foundation for moving forward. It allows us to establish clear expectations, and goals moving forward. We aim to fully complete Iteration 1 by March 6, and begin coding by March 7th. This allows us to get a head start on the tasks at hand and allows us to have sufficient time to understand our implementations. Furthermore, it gives us more time to start our Iteration 2 group work and allows us to comfortably finalize our roles in the group. Starting early, and devising a plan to break up into sub-teams for Iteration 2 labour should be done realistically before the Iteration period begins, to ensure we all have a plan and a clear idea on how we will go about things by the time we receive the work from other groups. Getting quality work done for Iteration 2 as soon as possible will be ideal, as it provides us with more time to work on the tasks due at the end of Iteration 3. More time allows us to implement the highest quality system, and will allow us to have more time debugging, and identifying potential issues which stands between us and success.

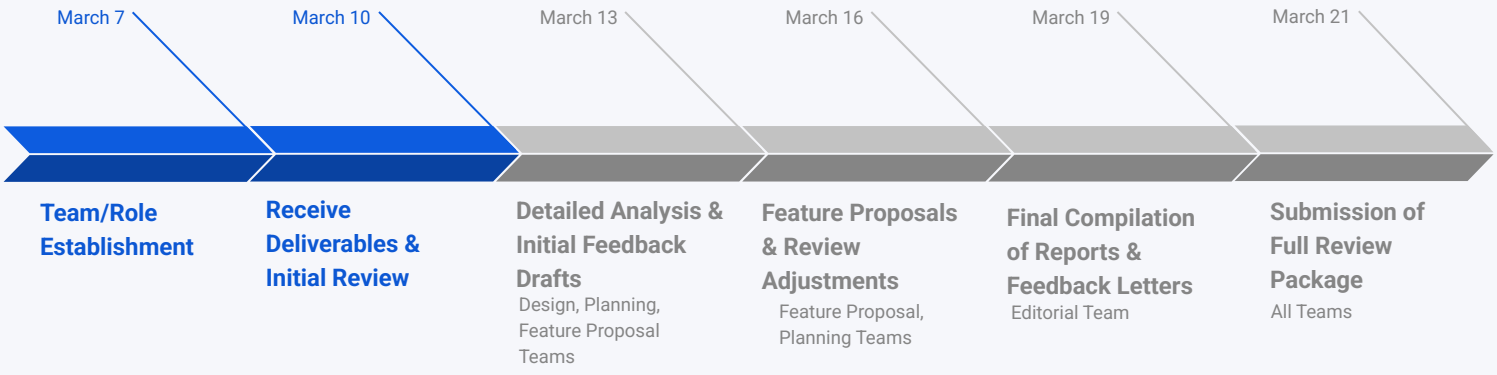
ITERATION 2 PLANNING:



Iteration 2 Group Tasks

Team	Members
Design Review Sub Team	6-8
Planning Analysis Sub Team	4-6
Feature Proposal Sub Team	6-8
Editorial / Documentation Sub Team	2-4

The teams highlighted above should look to be established no later than March 7, as each group should have communicated their expectations before going into the task. This will allow each group to begin right away. This also allows all 25 members to have a clear idea on ALL of their roles throughout the assignment, whether it relates to I1 / I3 Implementation, or just I2, and will allow them to swiftly complete their respective accountability contract at the start of I2. These contracts will act as another layback, allowing us to evaluate our options, and approaches to the tasks ahead.



Iteration 3 Group Tasks

ITERATION 3 PLANNING:

April 1	April 3	April 5	April 7
Team Realignment & Feedback Review - Review feedback from P2. - Discuss major design changes. - Assign team responsibilities for P3	**Implementation Start** - Begin implementing changes from P2 review. - Game Logic updates. - Start coding core missing functionalities.	**Component Development** - Authentication system refinements. - GUI updates for usability. - Initial networking integrations.	**First Integration & Debugging** - Merge updated submodules. - Run basic integration tests. - Identify key issues in early testing.

April 9	April 11	April 13	April 15
Full System Assembly & Refinements - Connect all modules. - Test interactions between matchmaking, networking, and leaderboard. - Start refining database interactions.	**Final Testing & Feature Halt** - Complete debugging. - Lock core features. - Validate leaderboard & game tracking.	**Documentation & Demonstration Prep** - Create final README and user guides. - Record demonstration videos. - Ensure proper GitLab documentation	**Final Submission & Wrap-Up** - Submit all required files. - Conduct final team reflections. - Ensure all members have their peer evaluations completed.

Leaderboard Sub-Team Planning

Iteration 1:

- Defining a structured stats system using an abstract parent class (GeneralStats) and game-specific child classes (ChessStats, GoStats, etc.).
- Implementing a ranking system with a universal ranking structure while allowing per-game MMR scaling.
- Designing a matchmaking system that efficiently pairs players using a Gaussian-distributed queue structure.
- Creating a leaderboard system that tracks and sorts player stats efficiently.
- Ensuring data persistence (simulated via a CSV file for leaderboard storage).

System Architecture

General Stats & Game-Specific Stats Classes:

- General Stats: Abstract class containing universal board game statistics.
- Game-specific classes (ChessStats, GoStats) that extend GeneralStats.

Matchmaking System:

- Uses an MMR-based Gaussian queue distribution for optimal pairings.
- Ensures players compete with similarly skilled opponents.

Ranking System:

- Uses universal ranking tiers with game-specific MMR ranges.
- Ranks are stored as an Enum for easy classification.

Leaderboard System:

- Stores player stats in a CSV-based database.
- Allows sorting by MMR or win count.

Potential Code Implementations

3.1 GeneralStats (Abstract Parent Class)

The GeneralStats class defines universal statistics shared across all board games.

Responsibilities:

- Track **wins, losses, ties, games played**.
- Provide **generic .win(), .lose(), .tie() methods** that update stats.
- Allow child classes to define **custom MMR updates**.

Potential Code Implementation:

```
from abc import ABC, abstractmethod

class GeneralStats(ABC):
    def __init__(self, player_id):
        self.player_id = player_id  # Unique identifier for the player
        self.wins = 0
        self.losses = 0
        self.ties = 0
        self.games_played = 0
        self.mmr = 100  # Default MMR for all games (child classes override this)

    def win(self):
        self.wins += 1
        self.games_played += 1
        self.update_mmr(win=True)

    def lose(self):
        self.losses += 1
        self.games_played += 1
        self.update_mmr(win=False)

    def tie(self):
        self.ties += 1
        self.games_played += 1

    @abstractmethod
    def update_mmr(self, win):
        pass  # Each game implements its own MMR scaling
        self._update_rank()  # Automatically updates rank after MMR change

    def _update_rank(self):
        """ Private method to update rank based on MMR. """
        rank_thresholds = [0, 30, 60, 90, 120, 150, 180, 200]  # Example MMR ranges
```

Potential Code Implementations

3.2 Game-Specific Stats Classes

Each game-specific class extends GeneralStats, adding game-specific statistics and implementing **MMR logic**.

ChessStats Example:

```
class ChessStats(GeneralStats):
    MAX_MMR = 200
    MIN_MMR = 0

    def __init__(self, player_id):
        super().__init__(player_id)
        self.moves_played = 0
        self.mmr = 100

    def record_move(self):
        self.moves_played += 1

    def update_mmr(self, win):
        if win:
            increment = (self.MAX_MMR - self.mmr) / 20
            self.mmr += increment
        else:
            decrement = (self.mmr - self.MIN_MMR) / 20
            self.mmr -= decrement
           self._update_rank()  # Automatically updates rank after MMR change

    def _update_rank(self):
        """ Private method to update rank based on MMR. """
        rank_thresholds = [0, 30, 60, 90, 120, 150, 180, 200]  # Example MMR ranges
        ranks = list(Rank)

        for i in range(len(rank_thresholds) - 1):
            if rank_thresholds[i] <= self.mmr < rank_thresholds[i + 1]:
                self.rank = ranks[i]
                break
```

Matchmaking System Planning

Matchmaking System

Matchmaking Approach:

- Uses MMR-based Ranked Queue Pairs instead of Gaussian distribution.
- Players are assigned to one of 14 rank-based matchmaking queue pairs (28 total queues).
- Each rank has 2 queue pairs (4 queues total per rank).
- Players are placed in one of the two queue pairs randomly to distribute load.
- Matching happens by pairing the two players at the front of a queue pair.
- If no one is in your queue pair after 1 minute, the player moves to the queue pairs of the rank below them to widen matchmaking without significantly affecting balance.

How the System Works:

1. Player Joins a Queue:
   - A player enters matchmaking and is assigned to one of two queues within their rank.
   - Example: A Diamond player goes into either Diamond Queue A or Diamond Queue B.

2. Matchmaking Process:
   - Every queue matches the first two players at the front.
   - If the queue is empty, the player waits for 1 minute.
   - After waiting, the player moves down to the queue pairs of the rank below them
   - Example: If a Diamond player waits too long, they will move to Platinum Queue A or B.

3. Avoiding Unfair Matches:
   - Players can only move down one rank at a time.
   - If a Platinum player also has no match, they move to Gold and so on.
   - A Grandmaster cannot drop below Master matchmaking queues to prevent mismatches.

4. How Players Are Matched
   - Two players at the front of a queue pair are matched together.
   - If there is only one player in a queue pair, they continue waiting or move down after 1 minute.

Matchmaking System

```
import random
import time

class MatchmakingSystem:
    def __init__(self):
        # 14 ranks, each has 2 queue pairs (so 28 queues total)
        self.queues = {rank: {0: [], 1: []} for rank in range(1, 15)}

    def add_player_to_queue(self, player):
        """Add a player to a random queue pair within their rank."""
        rank = player.get_rank().value
        queue_pair = random.choice([0, 1])
        self.queues[rank][queue_pair].append(player.player_id)

    def match_players(self):
        """Pair players in all rank-based queues."""
        matches = []
        for rank in self.queues.keys():
            for queue_pair in [0, 1]:  # Each rank has two queue pairs
                queue = self.queues[rank][queue_pair]
                while len(queue) >= 2:
                    player1 = queue.pop(0)
                    player2 = queue.pop(0)
                    matches.append((player1, player2))
        return matches

    def wait_and_move_down(self, player):
        """If a player waits too long, move them down to the next rank queue pair."""
        rank = player.get_rank().value
        if rank > 1:  # Players cannot go below rank 1
            time.sleep(60)  # Wait 1 minute
            new_rank = rank - 1
            queue_pair = random.choice([0, 1])
            self.queues[new_rank][queue_pair].append(player.player_id)

    def remove_player(self, player):
        """Remove a player from matchmaking if they leave."""
        rank = player.get_rank().value
        for queue_pair in [0, 1]:
            if player.player_id in self.queues[rank][queue_pair]:
                self.queues[rank][queue_pair].remove(player.player_id)
```

Ranking System Planning

Ranking System

The ranking system in the Leaderboard and Matchmaking System ensures that players are categorized into skill-based rank tiers, which remain consistent across all games while allowing flexibility in MMR scaling per game.

Purpose of the Ranking System:

- Provides a structured progression system that reflects player skill.
- Allows for game-specific MMR ranges while ensuring that rank distribution is evenly spaced within each game's MMR limits.
- Supports balanced matchmaking by ensuring players compete within reasonable MMR differences.
- Enables leaderboard sorting based on rank and MMR.

How the Ranking System Works Each game-specific stats class defines:

1. Minimum MMR (MIN_MMR) → The lowest possible rating.
2. Maximum MMR (MAX_MMR) → The highest possible rating.
3. Fixed Number of Ranks (7) → Ensures a consistent experience across all games.
4. MMR Steps per Rank → The range each rank covers, calculated dynamically

Calculating Ranks Dynamically Per Game:

Each game's Stats class will compute the rank boundaries based on its own MMR range. Formula for Each rank's MMR Step Each rank will span an equal portion of the game-specific MMR range:

$$\text{Rank Step} = \frac{\text{MAX\_MMR} - \text{MIN\_MMR}}{\text{Total Ranks}}$$

Ranking System Planning

Ranking System

```
class ChessStats(GeneralStats):
    MAX_MMR = 200  # Chess MMR cap
    MIN_MMR = 0    # Chess MMR minimum
    RANK_TIERS = 7  # Fixed number of ranks

    def __init__(self, player_id):
        super().__init__(player_id)
        self.moves_played = 0
        self.mmr = 100  # Default starting MMR


        def update_mmr(self, win):
            if win:
                increment = (self.MAX_MMR - self.mmr) / 20
                self.mmr += increment
            else:
                decrement = (self.mmr - self.MIN_MMR) / 20
                self.mmr -= decrement

        def get_rank(self):
            """Determine the player's rank based on their MMR."""
            step = (self.MAX_MMR - self.MIN_MMR) / self.RANK_TIERS
            rank_index = min(int(self.mmr // step), self.RANK_TIERS - 1)
            return Rank(rank_index + 1)
```

Ranking System

Rank Distribution for Different Games:

The fixed rank structure ensures fair matchmaking and consistent player experience while allowing each game to have different MMR scales

Game	Min MMR	Max MMR	MMR Step (Per Rank)
Chess	0	200	~28.5 MMR
Go	50	300	~35.7 MMR
Checkers	100	500	~57.1 MMR

Example Rank Assignments for Chess (0-200 MMR)

Ranking System

Rank	MMR Range
BRONZE	0 - 28 MMR
SILVER	29 - 57 MMR
GOLD	58 - 85 MMR
PLATINUM	86 - 114 MMR
DIAMOND	115 - 142 MMR
MASTER	143 - 171 MMR
GRANDMASTER	172 - 200 MMR

This method ensures that each game maintains a fair skill progression, regardless of how high or low its MMR values are.

How This Works With Matchmaking:

Players in the same rank are matched first, If no players in their rank are available, matchmaking extends slightly outside their MMR range while avoiding unfair matchups. This ensures high-ranked players do not face much lower-skilled opponents and vice versa.

Leaderboard System

Leaderboard System:

Functionality:

- Stores stats in a CSV file.
- Allows sorting by MMR or Wins.
- Loads & updates player stats efficiently.

Code Implementation:

```
import pandas as pd

class LeaderboardManager:
    def __init__(self, file_path="leaderboard.csv"):
        self.file_path = file_path
        self.data = self.load_data()

    def load_data(self):
        try:
            return pd.read_csv(self.file_path)
        except FileNotFoundError:
            return pd.DataFrame(columns=["Player ID", "Wins", "Losses", "MMR"])

    def save_data(self):
        self.data.to_csv(self.file_path, index=False)

    def update_player(self, player_id, wins, losses, mmr):
        if player_id in self.data["Player ID"].values:
            self.data.loc[self.data["Player ID"] == player_id, ["Wins", "Losses",
"MMR"]] = [wins, losses, mmr]
        else:
            new_row = pd.DataFrame([[player_id, wins, losses, mmr]], columns=["Player
ID", "Wins", "Losses", "MMR"])
            self.data = pd.concat([self.data, new_row], ignore_index=True)
        self.save_data()

    def get_top_players(self, by="MMR", top_n=10):
        return self.data.sort_values(by=by, ascending=False).head(top_n)
```

Player Class Planning

Player Class Overview

Player ID System:

The Player ID system serves as the unique identifier for each player within the Leaderboard and Matchmaking System. Every player's profile, stats, matchmaking history, and ranking progression are tied to this ID to maintain data consistency across different components

Responsibilities:

- Uniquely identifies each player within the system.
- Links player data across the Leaderboard, Matchmaking, and Game Statistics components.
- Prevents duplicate player entries by ensuring each Player ID is globally unique.
- Enables efficient retrieval and tracking of player-specific data.

Integration with Other Components:

- *Game Statistics:*
  - Each player's stats (e.g., Connect4Stats, TicTacToeStats, CheckersStats) are associated with their Player ID.
- *Matchmaking System:*
  - The system tracks players in matchmaking queues using their Player ID.
  - When matching players, the system queries their rank and MMR using the Player ID.
- *Leaderboard Manager:*
  - Uses Player ID as the primary key to track wins, losses, and MMR changes.
  - Ensures that ranking updates are assigned to the correct player profile.

Attributes:

+ playerId: String (Globally Unique Identifier)

Methods:

+ getPlayerId(): String → Returns the unique Player ID.
+ validatePlayerId(): boolean → Ensures validity and uniqueness within the system

Player ID Planning

Player Class Overview

The Player ID system is a core component that enables seamless data tracking and ensures every

```
public class Player {
    private final String playerId;  // Unique identifier
    private Connect4Stats connect4Stats;
    private TicTacToeStats ticTacToeStats;
    private CheckersStats checkersStats;

    // Constructor
    public Player() {
        this.playerId = generateUniqueId();
        this.connect4Stats = new Connect4Stats();
        this.ticTacToeStats = new TicTacToeStats();
        this.checkersStats = new CheckersStats();


    }

    // Generates a globally unique Player ID
    private String generateUniqueId() {
        return UUID.randomUUID().toString();
    }

    // Returns the player's unique ID
    public String getPlayerId() {
        return playerId;
    }

    // Retrieves game-specific stats
    public GeneralStats getStats(String gameType) {
        return switch (gameType.toLowerCase()) {
            case "connect4" -> connect4Stats;
            case "tictactoe" -> ticTacToeStats;
            case "checkers" -> checkersStats;
            default -> throw new IllegalArgumentException("Invalid game type");
        };
    }
}
```

Player Stats Planning

Class: "Player"

Class: "Player"

- **Fields:**
  - Username / playerID : String
  - Email : String
  - Password : String
  - Ranking : Rank (Enum)
  - connect4Stats : Connect4Stats
  - ticTacToeStats : TicTacToeStats
  - checkerStats : CheckerStats

- **Constructor → public Player (username)**
- **Methods :**
  - getStats(gameType : String) : GeneralStats
  - All the other getter and setter methods for the fields
  - updateUsername(old username, new username)
  - updateEmail(username, email)
  - updatePassword(username, password)

Player Stats Planning

Class: “PlayerStats”

```
public class PlayerStats {
    private Player player;

    public PlayerStats(Player player) {
        this.player = player;
    }

    public int getWinsForConnect4() { return player.getStats("Connect4").getWins(); }
    public int getLossesForConnect4() { return player.getStats("Connect4").getLosses(); }
    public int getTiesForConnect4() { return player.getStats("Connect4").getTies(); }
    public Rank getRankForConnect4() { return player.getStats("Connect4").getRank(); }

    public int getWinsForTicTacToe() { return player.getStats("TicTacToe").getWins(); }
    public int getLossesForTicTacToe() { return player.getStats("TicTacToe").getLosses(); }
    public int getTiesForTicTacToe() { return player.getStats("TicTacToe").getTies(); }
    public Rank getRankForTicTacToe() { return player.getStats("TicTacToe").getRank(); }

    public int getWinsForChecker() { return player.getStats("Checker").getWins(); }
    public int getLossesForChecker() { return player.getStats("Checker").getLosses(); }
    public int getTiesForChecker() { return player.getStats("Checker").getTies(); }
    public Rank getRankForChecker() { return player.getStats("Checker").getRank(); }
}
```

Login Page Planning

Class: “Login Page”

Class: “LoginPage”
//instance of LoginPage will be created by our main function with database as a parameter

- **ENUM CLASS** : State (Enum) //this is going to have a separate box diagram
  - UsernameTaken
  - EmailFormatWrong
  - VerificationCodeWrong
  - Success

- **Fields :**
  - database : CredentialsDatabase

- **Constructor** → LoginPage(CredentialsDatabase : database)
- **Methods**:
  - login(username, password) : HomePage
  - *(We will basically be doing a lookup in the database object like Player player = database.findPlayerByUsername(username) and then we will check if the player is not null (the username exists in the database and the password entered is the same as player.getPassword and everything is correct, we will create a new Homepage with the player and database as its parameter and return that, so basically the homepage will be associated to that player, or if not then we will return null. So the existence of homePage will determine if the login was successful or not. Something like:*
    HomePage home = loginPage.login("player1", "password123");
    if (home != null) { System.out.println("Login successful!");)

  - register (username, password, email) : ENUM
    (Just check if entered code is not empty for verification purpose)
  - forgot password (username) : boolean

Home Page Planning

Class: "HomePage"

Class: "HomePage"

- **Fields:**
  - player : Player
  - database : CredentialsDatabase
  - accountSettings : Settings

- **Constructor:**
  public HomePage(Player player, Database database)
  this.accountSettings = new Settings(player, database)
- **Methods:**
  - viewYourOwnRecords:
  - PlayerStats playerStats = new PlayerStats(player)

  - viewOtherPlayerRecords:
  - PlayerStats otherPlayerStats = new PlayerStats(otherPlayer)

  - viewSettings : return accountSettings of type Settings

  - playConnect4() : void
  - playTicTacToe() : void
  - playCheckers() : void

Settings Planning

Class: "Settings"

Class: Settings
Fields:

- Player
- Database

Constructor:

public Settings (Player player, CredentialsDatabase)
this.player = player, this.data = database

Methods:

- deleteAccount(password): boolean
- changeUsername(newUsername) : boolean
- changeEmail(newEmail) : boolean
- *(Just check if the player has entered a non-empty verif. code)*
- changePassword(oldPassword, newPassword): boolean
- logout : boolean

(Implementation : We will be saving data from the hashmap database object to the database.txt and then logout. More to be discussed on how to reflect the changes in the hashmap database object, is it our team doing it? Whenever a player wins a match or their ranking changes, who will be updating that change to the credentials database hashmap)

Settings Planning

Class: “Networking”

Fields:

- **connections** : List<Player>
- **serverStatus** : boolean

Constructor:

- **Networking()**
- Initializes the server connection and player list.

Methods:

- **handleConnections()** : void
- **syncPlayers()** : void
- **verifyServerStatus()** : boolean
- **sendPlayerActions(player: Player, action: String)** : void
- **disconnectPlayer(player: Player)** : boolean
- **reconnectToGame(player: Player)** : boolean
- **keepPlayersSynchronized()** : void
- **getGamesCatalogue()** : List<Game>

Class: “networkMatchmaking

Fields:

- **activeSessions** : List<GameSession>
- **database** : Database

Constructor:

- **GameSessionManager(Database database)**

Methods:

- **startSession(players: List<Player>, gameType: Game)** : GameSession
- **endSession(session: GameSession)** : void

Settings Planning

Class: “GameSessionManager”

Fields:

- **queue** : List<Player>
- **activeMatches** : List<Match>

Constructor:

- **Matchmaking()**

Methods:

- **findMatch(player: Player)** : Match
- **joinMatch(player: Player, match: Match)** : boolean
- **joinMatchAsParty(players: List<Player>)** : Match

Class: “PartySystem

Class: "PartySystem"

Fields:

- **parties** : List<Party>

Constructor:

- **PartySystem()**

Methods:

- **createParty(leader: Player, members: List<Player>)** : Party
- **joinAsParty(party: Party)** : boolean

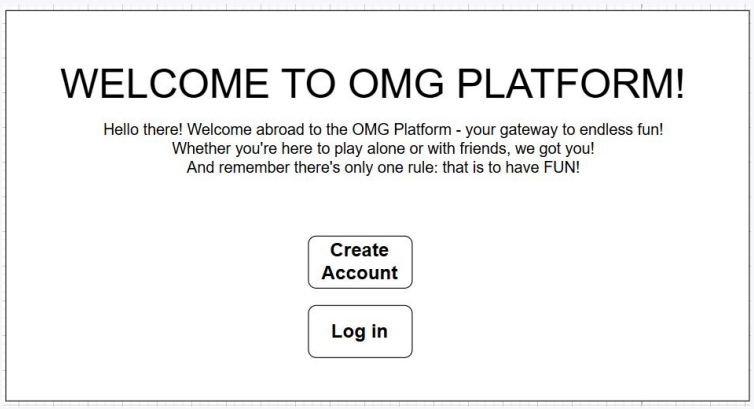
GUI Planning

Iteration 1:

- Defining user case descriptions for the game GUI to outline user interactions and system functionalities.
- Visually represent GUI interactions through user case diagrams.
- Developed structure diagram to illustrate the layout of the GUI framework.
- Creating a Low-fi Mockup system for GUI, for users to interact with the game boards, such as move pieces, display game stats, and chat with opponents,
- Working with the Game-Logic team to configure game expectations, interface implementation, and methods, attributes that are needed.

GUI Mock-Up

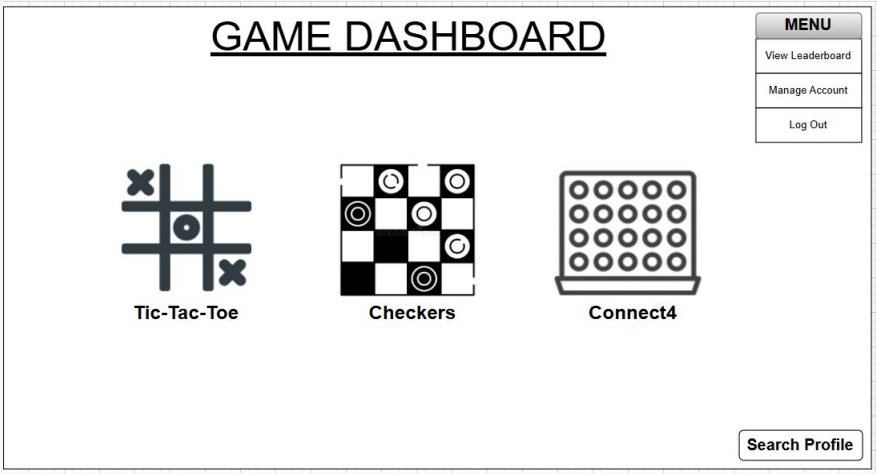
Welcome Page



This is the welcome page, where the players will be greeted onto the OMG Platform and given an option to either log in (if they have an existing account) or create an account.

- GridPane: for organizing elements, allowing for a structured layout
- TextFields: for adding text onto the interface
- Buttons: for interactions

Game Dashboard:

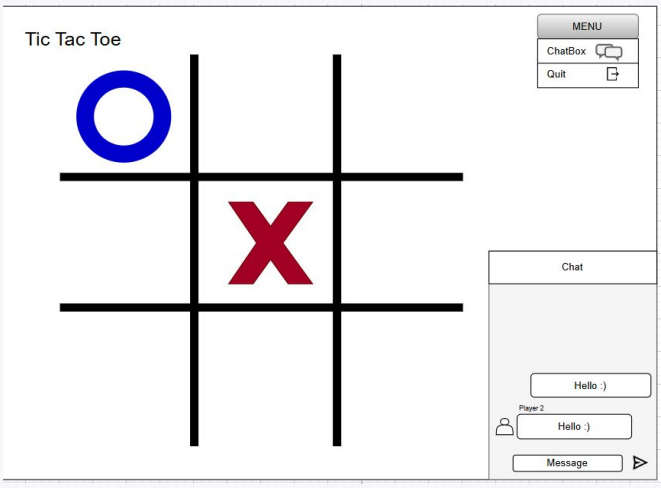


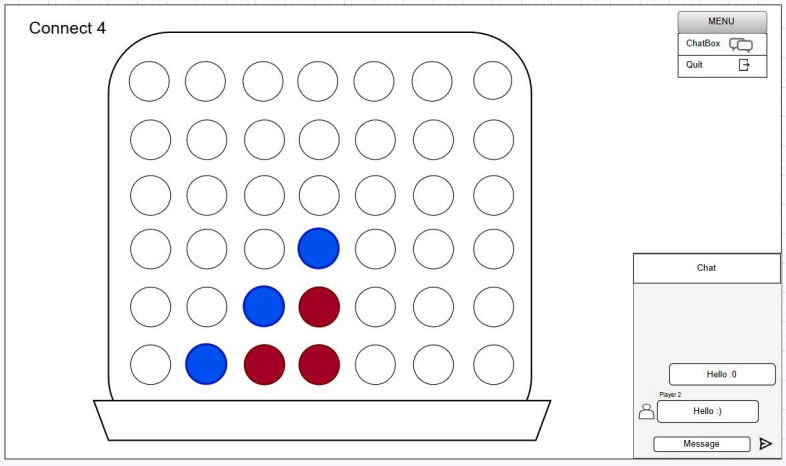
In order for us to implement this GUI planning to JavaFX, we would be using:

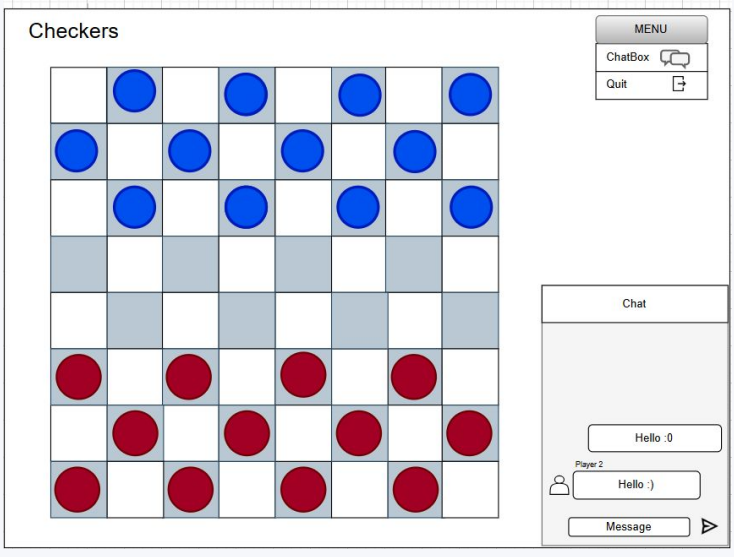
- GridPane: for organizing elements, allowing for a structured layout
- Buttons: for interactions
- ImageView: for icons

GUI Mock-Up

Game Pages



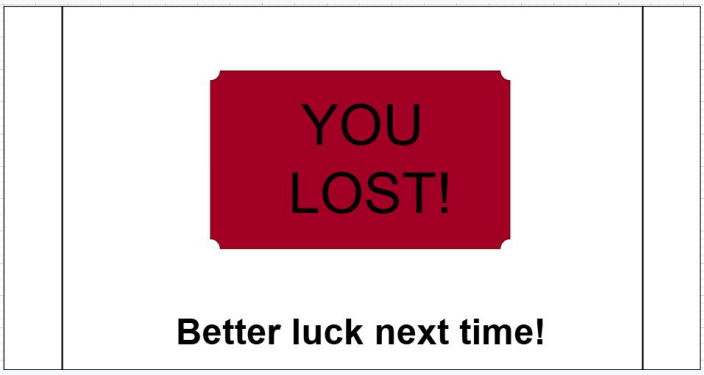




- Labels will be used for the title for each game (Checkers, Tic-Tac-Toe, Connect 4).
- GridePanes will be used to create a board.
    - Each cell will be a button to select the movement.
- ToggleButton will be used to open and close the ChatBox.
    - TextField to input the message.
    - TextArea to display the messages.
    - Button to send the message.
- Button to quit the game.
- ToggleButton for Menu option to show the available options (quit and chatBox)

GUI Mock-Up

Result Pages



- Label for the display texts (YOU WON!, YOU TIED!, YOU LOSE!)
- Label for the display descriptions (Amazing job!, Great effort!, Better luck next time!)
- ImageView for the illustrations on the pages.