# Leaderboard and Matchmaking System - Iteration 1

## 1. Overview

This document outlines the **Leaderboard and Matchmaking System** for **Iteration 1** of the SENG 300 project. The purpose of this system is to provide a structured ranking mechanism and an optimized matchmaking algorithm for board games, ensuring fairness and scalability.

This iteration focuses on:

- **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).

## 2. System Architecture

### 2.1 General Structure

The system is divided into **four key components**:

1. **GeneralStats & Game-Specific Stats Classes**
    - GeneralStats: Abstract class containing universal board game statistics.
    - Game-specific classes (ChessStats, GoStats) that extend GeneralStats.
2. **Matchmaking System**
    - Uses an **MMR-based Gaussian queue distribution** for optimal pairings.
    - Ensures players compete with similarly skilled opponents.
3. **Ranking System**
    - Uses **universal ranking tiers** with game-specific MMR ranges.
    - Ranks are stored as an Enum for easy classification.
4. **Leaderboard System**
    - Stores player stats in a **CSV-based database**.
    - Allows sorting by **MMR** or **win count**.

---

# 3. Class Design & Implementation

## 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
```

```
        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
```

---

## 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
```

---

## 3.3 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**.

```
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)
```

## 3.4 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.

**1. 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**.

---

## 2. Structure of the Ranking System

The ranking system consists of **seven fixed ranks**, represented using Python's `Enum` class:

```
from enum import Enum


class Rank(Enum):
    BRONZE = 1
    SILVER = 2
    GOLD = 3
    PLATINUM = 4
    DIAMOND = 5
    MASTER = 6
```

```
    GRANDMASTER = 7
```

These **seven ranks** are **universal** across all games, but the MMR **ranges for each rank** are dynamically adjusted **based on the min and max MMR caps of the game**.

---

## 3. 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.

---

## 4. 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**:

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

Using this step, we assign **rank tiers dynamically**.

**Example Implementation in a Game-Specific Class**

```
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)
```

---

## 5. Example: 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)**

| 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.

---

## 6. 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.

---

### 3.5 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)
```

---

# 4. 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.

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");
        };
    }
}
```

# 5. Planning Timeline for Project

| Week | Task |
|---|---|
| Week 1 | Define **use cases**, finalize system structure |
| Week 2 | Complete **class diagrams** and finalize data models |
| Week 3 | Implement core classes (GeneralStats, GameStats) |
| Week 4 | Implement MatchmakingSystem and LeaderboardManager |
| Week 5 | Conduct unit tests |
| Week 6 | Final debugging & documentation |

This document serves as the **foundation for iteration 1**, ensuring future expandability. 🚀

# 6. Notes

- All code is purely for a basic idea and abstract, not for real practical implementation