Cairo University

Faculty of Computers and Artificial Intelligence

2023/2024

OOP - CS213

Dr. Mohamed Elramly

Assignment 2: Games Report

|  |  |  |
| --- | --- | --- |
| Name | ID | Email |
| Adel ahmed Mohamed el hefny | 20230198 | [adelahmedhefny@gmail.com](mailto:adelahmedhefny@gmail.com) |
| Ahmed Mohamed Noureldin Ahmed | 20230035 | [ahmednourr278@gmail.com](mailto:ahmednourr278@gmail.com) |
| Ahmed Mohamed Mahmoud Ahmed | 20230598 | [ahmedabdellatif5355@gmail.com](mailto:ahmedabdellatif5355@gmail.com) |

* Who Did What!!!!!!

|  |  |
| --- | --- |
| **Name** | **Problems** |
| Adel Ahmed Mohamed el hefny | * Four-in-a-row * Numerical Tic Tac Toe * Ultimate Tic Tac Toe * 4×4 Tic Tac Toe |
| Ahmed Mohamed Noureldin Ahmed | * Pyramid Tic Tac Toe * Word Tic Tac Toe * Ultimate Tic Tac Toe * 4×4 Tic Tac Toe |
| Ahmed Mohamed Mahmoud Ahmed | * 5×5 Tic Tac Toe * Misere Tic Tac Toe * Ultimate Tic Tac Toe * 4×4 Tic Tac Toe |

* ***Description of Problems***
* Problem 1: Pyramid Tic Tac Toe

**Game Setup**

* The game is played on a **pyramid-shaped board** with 3 rows and 5 columns.
* Some positions on the board are **invalid**, giving the board its pyramid structure.  
  Valid positions include:
  + Row 0: (0,2)
  + Row 1: (1,1), (1,2), (1,3)
  + Row 2: (2,0), (2,1), (2,2), (2,3), (2,4)

**Gameplay**

* Two players take turns placing their symbols (e.g., X or O) on valid board positions.
* The game can be played by two human players or one human against a **random computer player** or **AI Player**.

**Draw Condition**

* The game ends in a draw if all 9 valid positions are filled and no winning condition is met.

**Winning Conditions**

* A player wins by forming a line of three identical symbols in:

1. A **row**:
   * E.g., (1,1), (1,2), (1,3) or (2,2), (2,3), (2,4).
2. A **column**:
   * E.g., (0,2), (1,2), (2,2).
3. A **diagonal**:
   * E.g., (0,2), (1,1), (2,0) or (0,2), (1,3), (2,4).

* Problem 2: Connect4\_board<T>

Purpose: Manages the game board for Connect4.

Key Methods:

* update\_board(): Updates the board with a player's move.
* display\_board(): Displays the current state of the board.
* is\_win(): Checks if there is a winner (4 consecutive symbols).
* is\_draw(): Checks if the game is a draw.
* game\_is\_over(): Determines if the game is over (win or draw).
* check\_direction(): Checks for consecutive symbols in a specific direction.

2. Connect4\_Player<T>

Purpose: Represents a human player in the game.

Key Method:

getmove(): Prompts the player to input a move.

3. Connect4\_Random\_Player<T>

Purpose: Represents an AI player that makes random moves.

Key Method:

getmove(): Randomly selects a column for the move.

Summary:

These classes implement the logic for a Connect4 game, allowing both human and random AI players to interact with the board, make moves, and check for a win or draw condition. The Connect4\_board class manages the game state, while Connect4\_Player and Connect4\_Random\_Player represent human and AI players, respectively.

* **Problem 3 : 5 x 5 Tic Tac Toe**

This tic-tac-toe variation is played on a 5 x 5 grid. As in the traditional game, players are Xs or Os.

* Rules:
* Players take turns placing an X or an O in one of the squares until all the squares except one are filled. (Each player has 12 turns for a total of 24 squares.)
* Winning:
* Count the number of three-in-a-rows each player has. Sequences can be vertically, horizontally, or diagonally. Whoever has the most, wins.
* Additional Rules:
* Decide ahead of time whether one mark can be counted in more than one three-in-a-row sequence. For easier implementation, allow counting more than once.
* Problem 4: Word Tic Tac Toe

**Game Setup**

* The game is played on a standard 3x3 board.
* Each player uses a letter of the alphabet (A-Z) as their symbol during the game.
* A dictionary file (dic.txt) is loaded at the start of the game, containing valid words.

**Gameplay**

* Players take turns placing their letter symbols on the board by choosing valid positions.
* The goal is to form valid 3-letter words horizontally, vertically, or diagonally.
* Words can be matched in **normal** or **reversed** order (e.g., "CAT" or "TAC").
* The game can be played between two human players or against a computer-controlled **random player** that makes moves and chooses letters randomly.

**Draw Condition**

* The game ends in a draw if all 9 board positions are filled and no valid word is formed.

**Winning Conditions:**

* A player wins by forming a valid 3-letter word (from the dictionary) in:

1. **Rows**:  
   Example: (0,0), (0,1), (0,2) forms a valid word.
2. **Columns**:  
   Example: (0,0), (1,0), (2,0) forms a valid word.
3. **Diagonals**:  
   Example: (0,0), (1,1), (2,2) forms a valid word.

Both forward and backward words are checked against the dictionary.

Problem 5: **Numerical Tic-Tac-Toe Game Classes Description**

**1. Numerical\_X\_O\_board<T> Class**

Overview  
The main game board class responsible for managing the game state, moves, and board logic. It is implemented as a template class for flexibility.

Key Features  
- 3x3 grid game board  
- Dynamic memory management  
- Move tracking  
- Win and draw condition checking

Important Methods

1. Constructor  
 - Initializes a 3x3 board with zero values  
 - Sets up initial game state

2. Destructor  
 - Properly frees dynamically allocated memory  
 - Prevents memory leaks

3. update\_board()  
 - Places a number on the board  
 - Validates move legality  
 - Tracks number of moves  
 - Manages number pool for used numbers

4. display\_board()  
 - Prints the current board state  
 - Shows cell coordinates and current values

5. is\_win()  
 - Checks for winning condition  
 - Verifies if any row, column, or diagonal sums to 15  
 - Ensures all three numbers are non-zero

6. is\_draw()  
 - Determines if the game is a draw  
 - Checks if 9 moves are completed without a winner

7. game\_is\_over()  
 - Combines win and draw checks  
 - Determines if the game has reached a final state

**2. Numerical\_X\_O\_player<T> Class**

Overview  
Represents a human player in the Numerical Tic-Tac-Toe game, managing player-specific game interactions.

Key Features  
- Player name and symbol management  
- Number pool tracking  
- Interactive move selection

Important Methods

1. Constructor  
 - Initializes player name and symbol  
 - Populates number pool with appropriate numbers  
 - Player 1 (odd symbol): 1, 3, 5, 7, 9  
 - Player 2 (even symbol): 2, 4, 6, 8

2. getmove()  
 - Interactively prompts player for move  
 - Allows board coordinate selection  
 - Enables number selection from available pool  
 - Validates number selection

**3. Numerical\_X\_O\_Random\_Player<T> Class**

### Overview  
An AI player that generates random moves and number selections.

Key Features  
- Automated move generation  
- Random coordinate and number selection  
- Adheres to game rules

Important Methods

1. Constructor  
 - Initializes number pool  
 - Ensures numbers match player's symbol parity

2. getmove()  
 - Randomly selects board coordinates  
 - Chooses a random number from available pool

**Game Mechanics**

Number Selection Rules

- Player 1: Uses odd numbers (1, 3, 5, 7, 9)

- Player 2: Uses even numbers (2, 4, 6, 8)

- Each number can be used only once

- Goal: Create a line summing to 15

**Winning Conditions**

- Three numbers in a line (row, column, diagonal)

- Line must total exactly 15

- All three numbers must be non-zero

**Game Flow**

1. Initialize game board

2. Players take alternating turns

3. Each turn involves:  
 - Selecting board coordinates  
 - Choosing an available number  
 - Updating board

4. Check for win or draw after each move

5. Game concludes when a player wins or board is full

**Strategic Considerations**

- Mathematically enhanced Tic-Tac-Toe

- Requires strategic number selection

- Adds complexity beyond traditional game

- Develops mathematical and strategic thinking

**Conclusion**

Numerical Tic-Tac-Toe transforms the classic game into a more intellectually challenging experience by introducing mathematical constraints and strategic number selection.

- Problem 6 : **. Misere Tic Tac Toe**

Misere Tic Tac Toe, also known as Inverse Tic Tac Toe or Toe Tac Tic, is a unique twist on the classic game. In this version, the objective is to avoid getting three marks in a row. The game flips the traditional win condition on its head, making every move a strategic decision to prevent losing.

* Rules:

The game is played on a standard 3x3 Tic-Tac-Toe grid. The goal is to avoid placing three of your marks in a row, column, or diagonal. The player who ends up with three marks in a row loses the game. If all cells are filled without either player aligning three marks in a row, the game ends in a draw.

Problem 7**. 4 x 4 Tic Tac Toe**

* 4 x 4 Tic Tac Toe is an extended version of the classic game, played on a larger board with more complex strategies. Each player has four tokens and aims to align three tokens in a row to win. This game introduces new movement rules and strategic depth to the traditional Tic-Tac-Toe.
* Rules:  
  - The game is played on a 4x4 grid.  
  - Each player has four tokens, placed in specific starting positions: two tokens on opposite sides of the board for each player.  
  - Players alternate turns, moving one of their tokens to an immediately adjacent open square. Tokens can be moved horizontally or vertically but not diagonally. Tokens may not jump over other tokens.
* Winning: The first player to get three tokens in a row wins the game. The alignment can be in any direction: horizontal, vertical, or diagonal.

**Problem 8. Ultimate Tic Tac Toe**

Ultimate Tic Tac Toe is an advanced version of the traditional game, adding layers of strategy and complexity. Players compete on a large 3x3 grid where each cell contains a smaller 3x3 Tic Tac Toe board.

**Rules:**

Player 1 starts by choosing any of the nine smaller Tic Tac Toe boards to play on.

Players alternate turns, playing Tic Tac Toe on the smaller boards.

The winner of a smaller board claims that space on the main board by replacing it with their symbol (X or O).

The next move is constrained to the smaller board that corresponds to the cell played in the previous move.

If a smaller board is already won or full, the next player can choose any available smaller board.

**Winning:**

The first player to win three smaller boards in a row, column, or diagonal on the main 3x3 grid wins the game.

Code reviews:

* **Ahmed Noureldin's review**
* Adel Hefny:
  + It's easy to read code & Not repeated.
  + The code is well-organized.
  + some comments describe obvious behavior (e.g., // Check in the positive direction, // Returns true if there is any winner) and **could be removed**.
  + Nested loops and conditions (e.g., in check\_direction() and simulate\_game()) are used judiciously but could be refactored further **to minimize complexity**.
  + **Some** code **violates DRY**(Don't Repeat Yourself) principles, like:
    - Board validation logic is repeated in multiple methods (e.g., update\_board, simulate\_game, get\_valid\_moves).
    - Directional checks in check\_direction(), I think it could be generalized into a single reusable method.
  + No exception handling is implemented, which could lead to runtime issues, especially with inputs.
* Ahmed Abdellateef:
  + The code is modular, well-organized, easy to read it.
  + Core game logic works, but Misere-specific rules are not implemented.
  + Readable and extensible, but is\_win could be refactored for efficiency.
  + No exception handling is implemented, which could lead to runtime issues, especially with inputs.
* **Adel hefny review**
  + **Ahmed Abdellateef:**
    - Strengths:

1. **Class Design:** Clear separation between the board, player, and random player classes.
2. Random Move Logic: The random move generation in TicTacToeRandomPlayer is correct and seeded to avoid repeating patterns.
   * + Areas for Improvement:
       1. Incomplete Win Check The is\_win() function is currently a placeholder. It should be implemented to check if a player has won the game by identifying sequences of three identical symbols in any row, column, or diagonal.
       2. Draw Check The is\_draw() function is not yet implemented. You can enhance this by checking if all cells are filled and there is no winne
       3. Score Calculation (get\_scores()):The get\_scores() function calculates the number of three-in-a-row sequences for each player. It would be more effective to track the winner dynamically in the is\_win() function rather than relying on counting sequences.
       4. Efficiency: The count\_three\_in\_a\_row() function checks for all possible three-in-a-row combinations, which could be optimized for larger boards or if the game’s win condition changes.
       5. Magic Number (24): In game\_is\_over(), the number 24 (max moves before the game is over) is hardcoded. It's better to derive this from the board's size to make the code more flexible.
       6. Unnecessary setName() in Player: The setName() method in TicTacToePlayer is trivial and doesn’t add significant value if the name is set only once during instantiation.

* **Ahmed Noureldin**
  + The code is a functional implementation of a word-based Tic-Tac-Toe game. It utilizes templates for flexibility and inheritance to separate concerns like the board and player logic.
  + The use of set<string> sett for storing valid words is a good approach, but it's important to handle file errors carefully.

**Suggestions for Improvement:**

* + Memory Management The destructor for Board\_Word cleans up the board correctly, but it can be optimized by using std::vector instead of dynamic memory allocation (new/delete[]). This would eliminate the need for manual memory management, making the code safer and more maintainable.
  + Error Handling: In readDic(), if the dictionary file doesn't open, the program prints an error and returns. It might be better to handle this case more gracefully, perhaps by terminating the program or providing a retry mechanism.
  + Game Logic: The is\_win() function checks for wins by examining rows, columns, and diagonals. This works fine, but the repetitive code can be simplified by creating a helper function that checks if a line (row/column/diagonal) matches any valid word in sett.
  + getmove() Method in Player\_Word: • The method asks the user for both the move coordinates and the character. It's functional but could be separated into two functions: one for getting the coordinates and one for getting the symbol. This would allow for more flexibility and clearer separation of concerns.
  + Code Readability: Make better use of whitespace and consistent indentation for readability.
  + Try adding comments to clarify logic, especially in complex functions like is\_win().

Suggeste Refactore is\_win() for DRY (Don’t Repeat Yourself) Principle

* Summary:

Your code is well-organized, but there's room for improvement in terms of error handling, memory management, and simplifying the logic. Using std::vector instead of raw arrays would make the code cleaner and safer. Consider refactoring the win-checking logic and separating concerns more clearly in the getmove() method.

Ahmed Abdellatif review:

* Ahmed Noureldin:

The header files implement different board games using the generic framework provided in BoardGame\_Classes.h. They include Pyramid\_XO.h, Pyramid\_XO\_AI.h, and Word\_TicTacToe.h. Each file has unique implementations while sharing a consistent structure. Here's the review:

**1. Requirements Compliance**

The files meet their intended purpose, extending the framework to specific games (Pyramid XO, Pyramid AI, and Word Tic-Tac-Toe).

They correctly inherit from the generic Board, Player, and RandomPlayer classes, adding game-specific logic such as custom board configurations, invalid positions, and win conditions.

**2. Code Formatting**

**Strengths**:

Consistent use of indentation and braces throughout the files.

Variable naming is generally meaningful (x, y for coordinates, symbol, etc.).

Use of templates (template<typename T>) aligns with the generic design.

**Areas to Improve**:

Avoid using #include <bits/stdc++.h>. While convenient, it includes unnecessary headers, increasing compilation time. Use specific headers instead (e.g., <iostream>, <fstream>, <set>, <string>, etc.).

Avoid large commented sections (///////------Implementation-------///////). Replace them with concise function or class documentation.

**3. Best Practices**

**Strengths**:

Usage of STL containers like set for invalid positions and dictionary words is appropriate and efficient.

The Minimax algorithm in PyramidAI follows a clear logic structure.

**Areas to Improve**:

**Global variables**:

set<pair<int, int>> s (invalid positions in Pyramid\_XO) and invalidPos (in Pyramid\_XO\_AI) should not be global. Consider moving them into the relevant classes as static members or instance variables.

**Magic Numbers**:

Numbers like 3 (rows), 5 (columns), and 9 (total moves) are hardcoded. Define them as constants or use parameters for better readability and flexibility.

**Avoid Hardcoding Paths**:

The file path "dic.txt" in Word\_TicTacToe.h should be parameterized to make the implementation more adaptable.

**Memory Management**:

Board classes dynamically allocate memory for the 2D board (this->board), but there’s no explicit destructor to release it. Add destructors to prevent memory leaks.

**4. Maintainability**

**Strengths**:

Code structure is modular, with clear separation between Board, Player, and specific game logic.

Template usage enhances reusability.

**Areas to Improve**:

**Repetition in Win Conditions**:

The win-checking logic in all Board classes is repetitive and prone to errors when changes are needed. Refactor it into reusable helper functions to reduce duplication.

For example, checking rows, columns, and diagonals can be encapsulated in a method like checkLine.

**5. Performance**

**Strengths**:

Efficient use of data structures like set for constant-time lookups.

The Minimax implementation in PyramidAI is well-structured for decision-making.

**Areas to Improve**:

**Minimax Optimization**:

The calculateMinMax function in PyramidAI lacks pruning (e.g., alpha-beta pruning). Adding this would significantly improve performance for larger boards or deeper searches.

**Unnecessary Copies**:

Avoid repeated copying of strings (e.g., word2 = word; reverse(word2.begin(), word2.end());). Perform these operations in-place when possible.

**6. Architecture**

**Strengths**:

The files extend the base framework appropriately, maintaining consistency in interface and implementation.

The use of polymorphism (e.g., overriding getmove() in different player types) aligns with the framework's design.

**Areas to Improve**:

Tight Coupling with Board:

In PyramidAI, calculateMinMax directly interacts with this->boardPtr. Consider abstracting this interaction into higher-level methods to reduce dependency.

**7. Testing**

**Strengths**:

The design supports modular testing (e.g., testing is\_win() independently of other methods).

**Areas to Improve**:

**Edge Cases**:

Ensure the handling of invalid moves (e.g., out-of-bounds or in invalid positions) is tested thoroughly.

For Word\_TicTacToe, test cases for non-standard dictionaries or uppercase/lowercase mismatches should be included.

**Unit Tests**:

There’s no mention of test coverage. Implement unit tests for each critical function, particularly update\_board, is\_win, and getmove.

**8. Documentation**

**Strengths**:

Class names and function names are self-explanatory (e.g., RandomPlayer\_Pyramid, is\_draw, etc.).

**Areas to Improve**:

Add comments to explain non-obvious logic (e.g., why certain positions are invalid in Pyramid\_XO).

Provide clear documentation for public APIs, especially for template-based classes, as they might be less intuitive for new developers.

Explain the expected format of the dictionary file (dic.txt) in Word\_TicTacToe.

**Summary of Recommendations**

1. Refactor repetitive logic in win conditions into reusable helper functions.
2. Replace #include <bits/stdc++.h> with specific headers.
3. Avoid global variables (s and invalidPos), and define them within relevant classes.
4. Add destructors for memory cleanup to prevent leaks.
5. Optimize Minimax with alpha-beta pruning for better performance.
6. Parameterize hardcoded values (e.g., 3, 5, "dic.txt").
7. Improve documentation, especially for edge cases and expected inputs.
8. Add comprehensive unit tests and ensure edge cases are covered.

Adel Hefny :

**1. Requirements**

* **Does the code meet the specified requirements?**
  + The header files provide the functionality for a Connect 4 game, including classes for the board (Connect4\_board), human player (Connect4\_Player), random player (Connect4\_Random\_Player), and MCTS-based AI player (Connect4\_MCTSPlayer). These align with the requirements of a Connect 4 implementation.
  + Additional features like MCTS for AI-based decisions are implemented, going beyond basic requirements.
* **Is the code complete for the described use case?**
  + The code includes all necessary elements: board management, player input, win/draw checking, random moves, and AI simulation.

**2. Code Formatting**

* **Are naming conventions consistent and meaningful?**
  + Class names and method names follow a clear and meaningful pattern (e.g., Connect4\_board, update\_board, is\_win).
  + A minor issue is the inconsistency in method capitalization. For instance, getmove could be renamed to getMove for consistency.
* **Is the indentation and spacing appropriate?**
  + The indentation is mostly clear and consistent.
  + There is minor redundancy in #include statements (e.g., <cmath>, <memory> are included multiple times).
* **Are there redundant or unnecessary comments?**
  + Comments explaining standard logic (like for loop functionality) could be reduced to make the code cleaner.
  + Explanatory comments for complex logic (like MCTS) are useful and appropriate.

**3. Best Practices**

* **Is the code free of hard-coded "magic values"?**
  + The code contains hard-coded values like 6 for rows and 7 for columns in multiple places. These should be replaced with constants for maintainability (e.g., constexpr int ROWS = 6; and constexpr int COLUMNS = 7;).
* **Are exceptions and errors handled gracefully?**
  + Errors like out-of-range indices are handled with std::out\_of\_range.
  + In Connect4\_MCTSPlayer::get\_valid\_moves, there's an error message logged if no valid moves are found, but no graceful handling of this situation. A fallback mechanism should be added to prevent potential crashes.
* **Is the use of dynamic memory safe and efficient?**
  + The destructor in Connect4\_board correctly cleans up dynamically allocated memory.
  + Prefer using std::vector<std::vector<T>> instead of raw pointers (T\*\*) for the board to avoid manual memory management and reduce risks of leaks.
* **Is code duplication minimized?**
  + check\_direction is implemented twice (once in Connect4\_board and once in MCTS logic). Consolidating it into a shared utility function would reduce duplication.

**4. Maintainability**

* **Does the code follow the Single Responsibility Principle?**
  + Each class has a single responsibility: the board handles the game state, players make moves, and MCTS handles AI logic.
* **Is the code modular and reusable?**
  + The modular design makes it easy to extend or replace components (e.g., adding a new player type).
  + Some methods, like simulate\_game, are tied to Connect 4 specifics (e.g., hard-coded board size). Adding generalization would improve reusability.
* **Are dependencies minimal and well-managed?**
  + Too many dependencies are included at the top of the file (<cmath>, <random>, <memory>, etc.), and not all are necessary. Remove unused headers.

**5. Performance**

* **Does the code avoid unnecessary computation or memory allocation?**
  + expand\_node efficiently generates new states for MCTS by copying the board state.
  + The board uses raw pointers (T\*\*) with frequent manual memory access, which may degrade performance compared to modern STL containers.
* **Are data structures appropriate for the use case?**
  + Raw pointers for the board should be replaced with std::vector for better performance and safety.
* **Does the AI simulation scale well?**
  + MCTS is a computationally intensive algorithm, but its iterative nature makes it flexible for scaling based on available resources.

**6. Architecture**

* **Is the code structured logically?**
  + The overall structure is logical, separating game logic (Connect4\_board) from player behavior (Connect4\_Player and Connect4\_MCTSPlayer).
* **Are templates appropriately used?**
  + The use of templates in Board and Player classes makes the framework flexible for different data types.
* **Is the MCTS integration cohesive?**
  + MCTS logic is well-encapsulated in Connect4\_MCTSPlayer, adhering to separation of concerns.

**7. Testing**

* **Are edge cases handled?**
  + Some edge cases need further consideration:
    - What happens if a user provides invalid input in getmove?
    - How does the game handle a scenario where no valid moves are left during MCTS simulation?
* **Is there support for automated testing?**
  + The code does not explicitly support unit testing. Adding test cases for board updates, win conditions, and AI logic would improve reliability.
* **Are error conditions verified?**
  + Errors like out-of-bounds access are checked, but additional testing is needed for unexpected user input.

**8. Documentation**

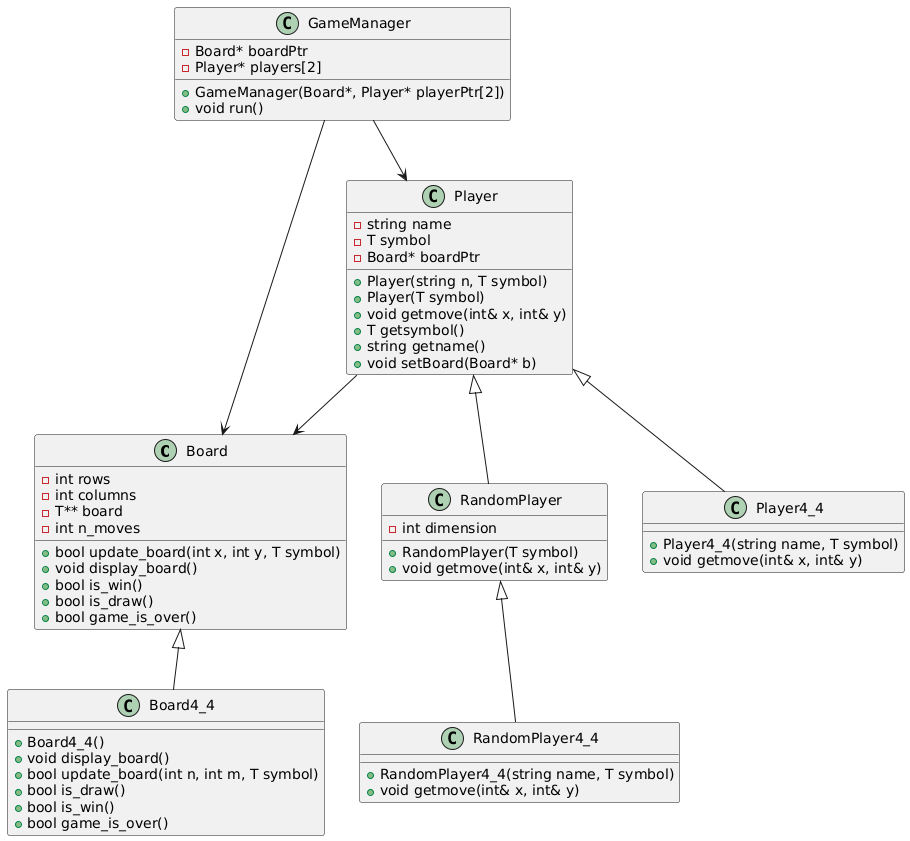
* **Are functions and classes adequately documented?**
  + Most functions lack descriptive comments explaining their purpose, inputs, and outputs. Adding Doxygen-style comments would improve readability.
* **Is the code self-explanatory?**
  + The code is generally understandable due to meaningful naming conventions, but some complex methods (e.g., simulate\_game) could use additional clarification.
* **Is external documentation provided?**
  + There’s no mention of external documentation for setting up or running the game. Including a README file or comments with usage instructions would help new users.

**Actionable Recommendations**

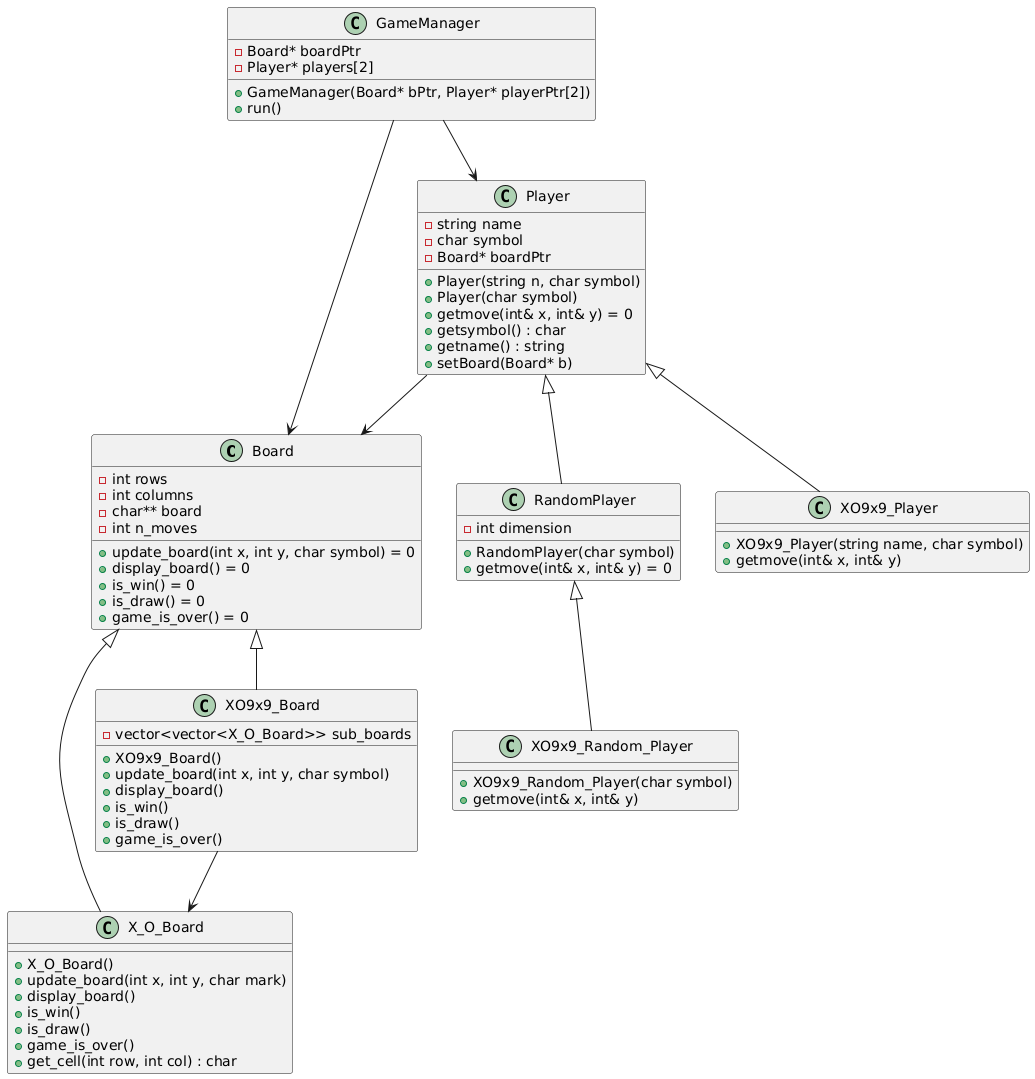
1. **Refactor for Maintainability:**
   * Replace raw pointers (T\*\*) with std::vector<std::vector<T>> in Connect4\_board.
   * Consolidate check\_direction into a shared utility function to reduce duplication.
2. **Replace Magic Values:**
   * Use constants (constexpr int ROWS = 6;, COLUMNS = 7;) instead of hard-coded numbers.
3. **Improve Error Handling:**
   * Add fallback mechanisms for cases where no valid moves exist in MCTS logic or user inputs invalid data.
4. **Optimize Dependencies:**
   * Remove unnecessary #include statements.
5. **Enhance Documentation:**
   * Add function-level comments to describe the purpose and behavior of methods.
   * Include a README or usage guide for setting up and running the game.

UML for group games :

Problem7 UML



Problem 8 UML



GitHub screenshot

