Faculty of Computers and

Artificial Intelligence - Cairo University

Name of the assignment: Assignment #2 - Task 2,3,4 - PDF Report – Prof. Mohamed El Ramly

Course: CS213 - Object Oriented Programming

Section: S7-8

Name1: Fahd Deya El-din

ID1: 20230287

E-mail1: fahd.cu2004@gmail.com

Name2:Mariam Hesham

ID2: 20230400

E-mail2: mariamheshammahdy147@gmail.com

Name3: Habiba Hany

ID3: 20230120

E-mail3: habibahanyramzi689@gmail.com

Introduction

This report provides an overview of the class architecture for a gaming framework. The system uses an object-oriented design to represent different types of boards, players, and gameplay strategies.

The design ensures modularity, extensibility, and clarity by utilizing inheritance and polymorphism.

Below is a detailed description of each class, its methods, and its functionality.

Class Descriptions

Board Class (Abstract Template)

The `Board` class is an abstract template class that defines the base functionality for different types of game boards. It includes the following attributes and methods:

- **Attributes**:

- `rows` and `columns`: Dimensions of the board.

- `board`: A 2D array representing the board state.

- `n_moves`: Counter for the number of moves made.

- **Methods**:

- `update_board(int x, int y, T symbol)`: Pure virtual method to update the board with a symbol.

- `display_board()`: Pure virtual method to display the board state.

- `is_win()`: Pure virtual method to check if there is a winner.

- `is_draw()`: Pure virtual method to check if the game is a draw.

- `game_is_over()`: Pure virtual method to determine if the game is over.

Player Class (Abstract Template)

The `Player` class is an abstract template that represents a player in the game. It includes the following:

- **Attributes**:
- `name`: Name of the player (e.g., 'Player 1' or 'Computer').
- `symbol`: Symbol used by the player (e.g., 'X' or 'O').
- `boardPtr`: Pointer to the board on which the game is played.

- **Methods**:
- Two constructors:
- `Player(string n, T symbol)`: Initializes a player with a name and symbol.
- `Player(T symbol)`: Initializes a computer player with a symbol.
- `getmove(int& x, int& y)`: Pure virtual method for player move input.
- `getsymbol()`: Returns the player's symbol.
- `getname()`: Returns the player's name.
- `setBoard(Board<T>* b)`: Sets the board pointer.

RandomPlayer Class (Derived Template)

The `RandomPlayer` class is derived from `Player` and represents a computer player that generates random moves. It includes the following:

- **Attributes**:
- `dimension`: Dimension of the board (used for random move generation).

- **Methods**:

- Constructor: `RandomPlayer(T symbol)` : Initializes the player and passes the symbol to the parent class.

- `getmove(int& x, int& y)` : Pure virtual method to generate a random move.

GameManager Class

The `GameManager` class manages the flow of the game, coordinating the board and the players. It includes:

- **Attributes**:

- `boardPtr` : Pointer to the board used in the game.

- `players[2]` : Array of pointers to the two players participating in the game.

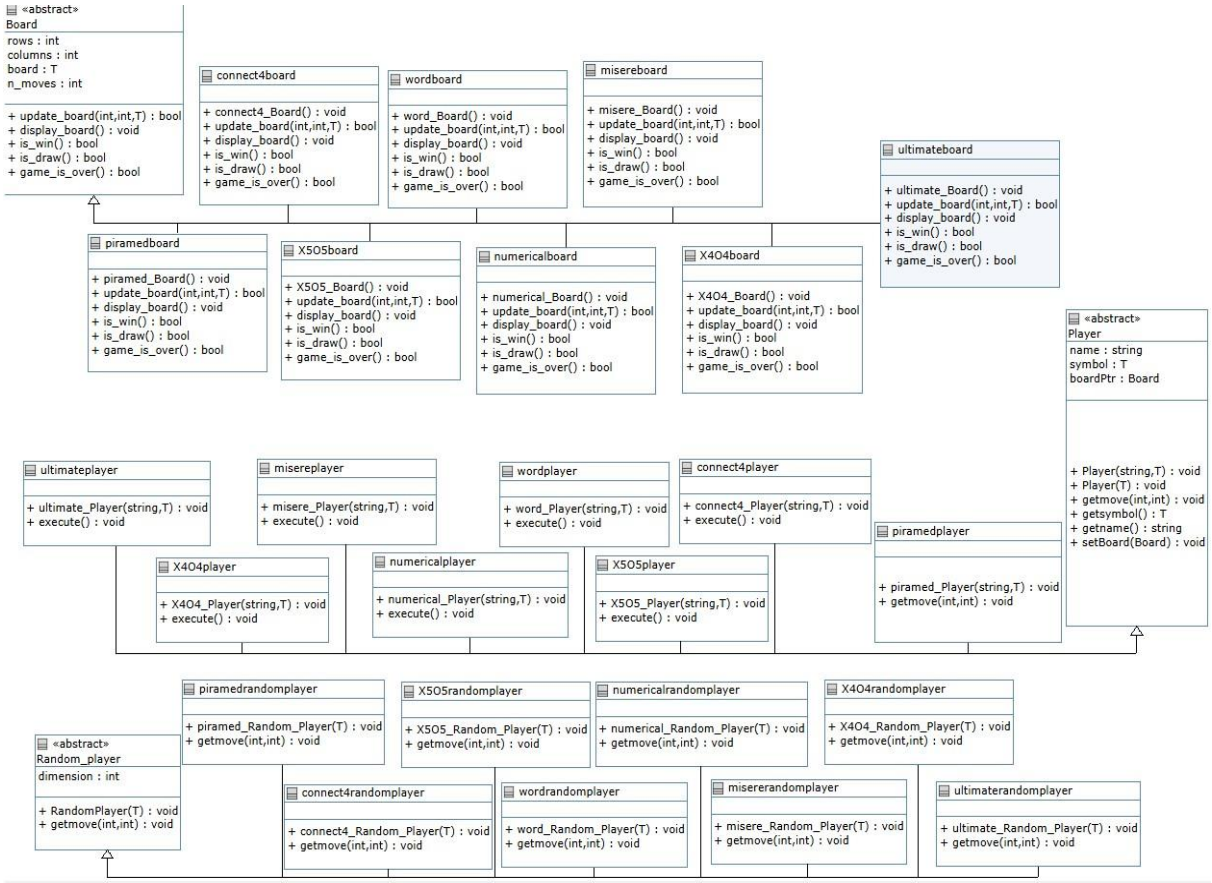
- **Methods**:

- Constructor: `GameManager(Board<T>* bPtr, Player<T>* playerPtr[2])` : Initializes the gamemanager with a board and two players.

- `run()` : Main game loop that alternates between players until the game is over, handling moves and checking for win/draw conditions.

UML

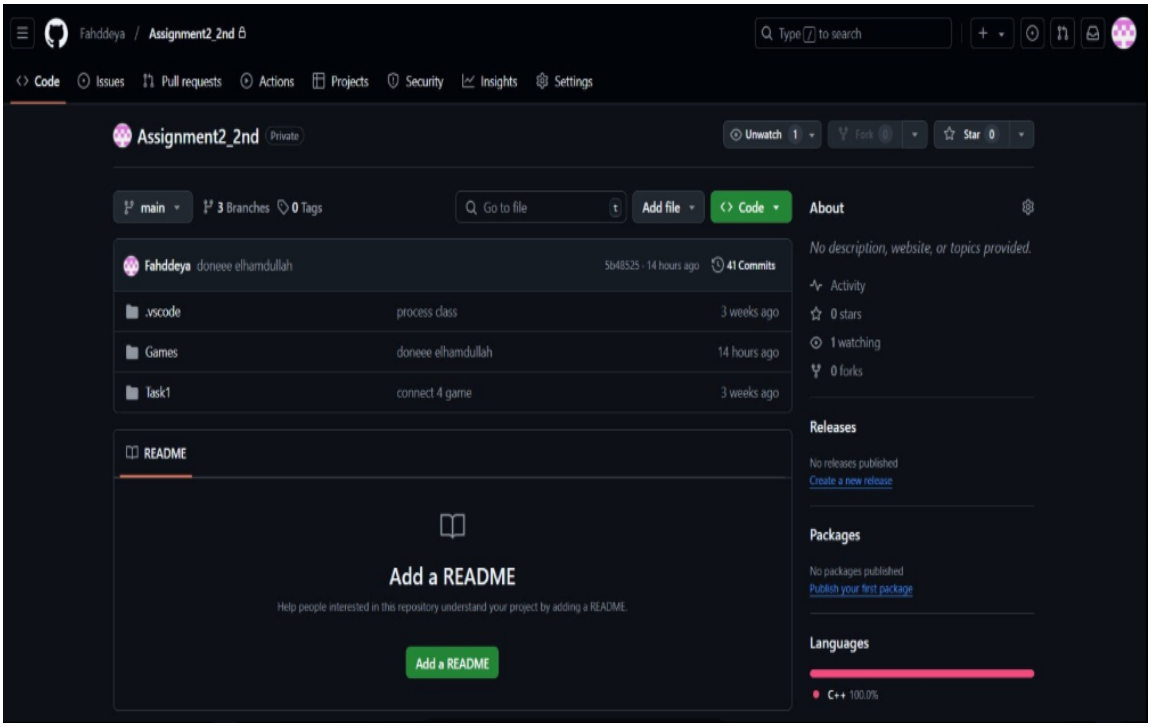
The diagram below illustrates the hierarchical relationships between the abstract and concrete classes. It highlights the inheritance structure, where specific game boards and players are derived from their respective abstract classes.



Break_Down Table :

Name	ID	Work Done
Mariam Hesham	20230400	Problem 3, 6, 8 and 9.
Habiba Hany	20230120	Problem 1, 4. The menu and the pdf report .
Fahd Deya	20230287	Problem 2, 5, 7 and 9.

Github Repo screenshot:



Pyramid Tic Tac Toe Game - Code Review

Code Quality:

- **Readability**: The code is easy to read, with clear naming conventions and function structure.
- **Single Responsibility**: Each function adheres to the single-responsibility principle.
- **Display Function**: The `display_board()` function effectively shows the current board state, though some formatting improvements could be made.
- **is_win() Function**: The win-check logic uses hardcoded conditions, which reduce maintainability.

Suggestions for Improvement:

1. **Refactor the `is_win()` Function**
   - Avoid hardcoded conditions for win checks; consider generalizing the logic to improve maintainability.
2. **Improve Board Display**
   - Standardize the spacing and formatting in the `display_board()` function to enhance readability.

Word Tic Tac Toe Game - Code Review

Code Quality:

- **Readability**: The code is readable with descriptive variable names and logical function breakdowns.
- **Single Responsibility**: Functions are clearly focused on a single task.

Suggestions for Improvement:

1. **AI Move Validation**
   - Modify the AI move generation to check for already occupied cells.
2. **Additional Comments**
   - Include comments for critical functions, particularly for the dictionary-loading and word-checking logic.

Ultimate Tic-Tac-Toe Game - Code Review

Code Quality:

- **Readability**: The code is clean and follows naming conventions, making it easy to understand.
- **Reusability**: Template classes enhance reusability for different games.

Suggestions for Improvement:

1. **Refactor `is_win()` Function**
   - The current win-checking logic uses hardcoded conditions; consider refactoring for more flexibility.
2. **Reduce Nesting in `update_board()`**
   - The logic for sub-board selection and initialization could be refactored into smaller helper functions to reduce excessive nesting.

Code Review for 4x4 Tic-Tac-Toe Implementation

Code Quality:

1. **Readability**
   - The code is modular with clear class and function names.
   - Variable names are clear, though `ca` and `cb` could be better named to reflect their role as current move coordinates.
2. **Reusability**
   - The use of template classes (`T`) allows for reusability across different types of game pieces.
3. **Error Handling**
   - Input validation prevents invalid moves.
4. **Board Display**
   - The `display_board()` method provides a detailed view of the board, including coordinates, making it clear to players where they can place their pieces.

Suggestions for Improvement:

1. **Improve `is_draw()` Logic**
    - The `is_draw()` function is currently incomplete and always returns false.
      Solution: Implement proper logic to check for a draw.
2. **Simplify `update_board()` Logic**
    - The `update_board()` function has excessive nested conditionals.
      Solution: Refactor to reduce complexity.
3. **Improve `getmove()` Logic for Player Input**
    - The input for selecting the move (`getcurrent()`) could be encapsulated in a loop that checks whether the chosen coordinates are valid and not occupied.
      Solution: Add a check to ensure the selected spot is empty before updating.
4. **Cleanup Resource Management**
    - Ensure that player and board objects are properly cleaned up, especially in cases of early returns or errors.

Code Review for 5x5 Tic-Tac-Toe Game

Code Quality:

1. **Readability**
    - The code is easy to follow and maintain. Functions are modular and adhere to single-responsibility principles.
    - Variable names are descriptive.
2. **Reusability**
    - The use of template classes provides flexibility, making the board and player types adaptable.
    - Functions like `check_wins()` are reusable for different game dimensions and board states, though they could be made more general.
3. **Formatting**
    - The board is displayed clearly with the use of `setw()` to ensure consistent spacing and formatting.
    - The `display_board()` function could be further optimized to handle empty and filled spaces uniformly.

Suggestions for Improvement:

1. **Board Display Consistency**
    - The board display could be more consistent, especially when displaying empty spots and symbols.

Code Review for Connect 4 Game

Code Quality:

1. **Separation of Concerns**
   - Classes for the board, player, and random player are clearly defined, maintaining modularity.
2. **Board Representation**
   - The `Connect4_Board` class is responsible for managing the game state. It initializes a 6x7 grid, supports updates to the board, and checks for wins and draws.
   - Methods like `update_board()` and `display_board()` are well-suited for handling the game state and presenting it to the user.
3. **Code Readability**
   - The code is generally clear and easy to follow, with reasonable variable names and logical structure.

Suggestions for Improvement:

1. **Magic Numbers**
   - There are magic numbers such as 6 (rows), 7 (columns), and 42 (total number of moves).
     Solution: These numbers should be replaced with named constants to make the code more readable and easier to maintain.
2. **Redundant Checks in `update_board()`**
   - In the `update_board()` method, there is an unnecessary check for `mark == 0` after confirming that the position is empty.
     Solution: Remove the redundant check.
3. **Fixed `x` Value in `getmove()` for Players**
   - In both `Connect4_Player` and `Connect4_Random_Player`, the `x` coordinate is always set to 0.
     Solution: Allow dynamic selection of `x` coordinates.
4. **Column Full Check in `getmove()`**
   - In `Connect4_Player::getmove()`, the column is selected without checking if the column is full.
     Solution: Implement a check for full columns.

Code Review for Numerical Board Game Implementation

Code Quality:

1. Readability:

- The code is modular and well-structured, with clear method names and logical flow.

2. Reusability:

- Template classes ensure that the code can be reused for various types of data types and game configurations.

3. Board Display:

- The `display_board()` method offers an easy-to-read output of the current board state.

Suggestions for Improvement:

1. Improve is_draw() Logic:

- The `is_draw()` function currently checks for a draw by ensuring 9 moves are made and no winner is found. This logic is correct but could be clarified and made more consistent.

2. Simplify update_board() Logic:

- The `update_board()` method contains nested conditions that could be simplified to improve readability.

Code Review for MinMax Player Implementation

Code Quality:

1. MinMax Decision-Making:

- The `calculateMinMax()` method evaluates all possible board states recursively, assigning scores based on whether a move leads to a win or loss.

2. Winning and Blocking Moves:

- The AI prioritizes making a winning move, followed by blocking the opponent's winning move. This helps the AI play tactically.

3. Flexible Move Representation:

- The `numerical_MinMax_Player` class supports numerical moves, which adds flexibility to the gameplay.

Suggestions for Improvement:

1. Simplify calculateMinMax() Logic:

- The `calculateMinMax()` method could be optimized by breaking it into smaller helper functions for evaluating board states.

2. Additional Input Validation:

- Further input validation on the player side could help in preventing invalid moves.

3. Extend AI Difficulty Levels:

- Adding different difficulty levels for the AI (e.g., beginner, expert) could provide a more varied gaming experience.

Code Review: 3x3 ToeTactic Game Implementation

Code Quality:

1. Win and Draw Detection:

- The `is_win()` method correctly checks for winning conditions across rows, columns, and diagonals.
- The `is_draw()` method detects when the board is full without a winner, allowing for a draw.

2. Player Interaction:

- The `getmove()` method for both the human and random AI players correctly prompts for moves or generates random moves, respectively.

Suggestions for Improvement:

Simplify** update_board() **Logic:

- The `update_board()` method could be simplified by refactoring condition checks into a separate function for improved readability.