



Cairo University

**Faculty of Computers and
Artificial Intelligence**

CS213: Object Oriented Programming

Assignment 1

• Submitted by:

Members' names	Information
Walid Adel Mordy Rohyem	20231200 Section 28 khattabadel112005@gmail.com
Mustafa Ehab Mustafa Akl	20231167 Section 28 mostafaehabakl@gmail.com
Sara Ibrahim Mohamed	20230166 Section 27 Sara6357141@gmail.com

• Submitted to:

Dr. Mohamed Elramly

Table of contents

1. Class description created for each game:.....	1
2. Report for the quality of the codes:	9
3. Bounce games' UML class diagram:	11
4. Git Hub repository screen shot:	12
5. Team members' work break-down:	13
6. GitHub repository link:.....	13

Summary

In an age where technology and innovation intertwine, the timeless appeal of classic board games like Connect Four and Tic-Tac-Toe remains unparalleled. This project brings these nostalgic games to life in a digital realm, leveraging the power of C++ to create engaging, dynamic, and interactive gameplay experiences. With meticulous class design, robust algorithms, and a collaborative development process, this report explores the journey of implementing these games. From detailed UML diagrams to in-depth code quality assessments, it reflects the seamless blend of creativity and technical precision that drives this endeavor.

-
- **Class description created for each game.**
-

• **Pyramid Tic-Tac-Toe**

1. **Pyramid_Board<T>**

- **Purpose:**
Represents the unique pyramid-shaped game board for Pyramid Tic-Tac-Toe. This board is triangular, with the topmost row containing 1 cell, the middle row 3 cells, and the bottom row 5 cells. The class handles the layout, updating moves, and checking win/draw conditions.
- **Attributes:**
 - board: A vector<vector<T>> that holds the current state of the game board, where T represents the type of the cell contents (e.g., 'X', 'O').
- **Key Methods:**
 - **Pyramid_Board():**
Constructor that initializes the pyramid layout:
Row 0: One cell at column 2.
Row 1: Three cells spanning columns 1 to 3.
Row 2: Five cells spanning columns 0 to 4.
 - display_board():**
Prints the board in a pyramid-like format with coordinates for each cell.
Shows valid moves as placeholders (e.g., .(x,y) if the cell is empty).
 - update_board(int x, int y, T symbol):**
Updates the board with a player's symbol if the move is valid and within the triangular structure.
Returns true if the update is successful, otherwise false.
 - is_win():**
Checks for a win condition:
Three consecutive symbols in a row.
Three consecutive symbols in a column.
Diagonal wins (from the apex to the base on both left and right sides).
 - is_draw():**
Returns true if all valid cells are filled and there is no winner.
 - game_is_over():**
Checks whether the game has ended by determining if there's a win or a draw.

2. **Pyramid_Player<T>**

- **Purpose:**
Represents a human player in Pyramid Tic-Tac-Toe. It allows players to make moves within the pyramid's constraints.
- **Attributes:**
 - Inherits from the Player<T> class, which includes:
 - name: The player's name.

- symbol: The player's mark (e.g., 'X' or 'O').
 - **Key Methods:**
 - **Pyramid_Player(const string& player_name, T player_symbol):**
 - Constructor that initializes the player's name and symbol.
 - **getmove(int& x, int& y):**
 - Prompts the player to enter their move (row x and column y).
 - Validates input to ensure it corresponds to a valid cell in the pyramid structure.
-

3. Pyramid_Random_Player<T>

- **Purpose:**
A computer-controlled player that makes random moves within the pyramid's valid structure.
 - **Attributes:**
 - Inherits from RandomPlayer<T>:
 - name: Defaulted to "Random Computer Player".
 - symbol: The computer's mark.
 - **Key Methods:**
 - **Pyramid_Random_Player(T player_symbol):**
 - Constructor that initializes the random player with a symbol.
 - Seeds the random number generator for generating moves.
 - **getmove(int& x, int& y):**
 - Randomly generates valid coordinates (x, y) based on the pyramid's structure:
 - Row 0: One cell (column 2).
 - Row 1: Three cells (columns 1-3).
 - Row 2: Five cells (columns 0-4).
 - Prints the chosen move.
-

• Connect Four Tic-Tac-Toe

1. Four_in_a_row_Board<T>

- **Purpose:**
Represents the game board for Connect Four, which is a 6x7 grid. This class manages the board layout, handles moves, and checks for win/draw conditions.
- **Attributes:**
 - rows: The number of rows (6 for Connect Four).
 - columns: The number of columns (7 for Connect Four).
 - board: A 2D array (pointer-based) to store the current state of each cell on the board.
 - n_moves: The number of moves made so far on the board.
- **Key Methods:**
 - **Four_in_a_row_Board():**
 - Constructor that initializes the board to 6 rows and 7 columns with all cells set to 0 (empty).
 - **~Four_in_a_row_Board():**
 - Destructor that deallocates the memory used by the dynamically allocated board.
 - **display_board():**
 - Displays the current state of the board in a formatted way, showing each column number at the top and a visual grid for the rows.

- **update_board(int x, int y, T mark):**
 - Updates the board at the specified (x, y) position with the given mark (player's symbol).
 - If the mark is 0, it resets the cell (used for undo functionality).
 - **is_win():**
 - Checks for a winning condition:
 - Horizontal, vertical, and diagonal checks to find if a player has aligned 4 of their symbols consecutively.
 - **is_draw():**
 - Returns true if the board is full (42 moves) and no winner has been found.
 - **game_is_over():**
 - Returns true if the game has ended either due to a win or a draw.
-

2. Four_in_a_row_player<T>

- **Purpose:**
Represents a human player in Connect Four. The player selects a column to drop their symbol and the class handles input validation.
 - **Attributes:**
 - Inherits from the Player<T> class:
 - name: The player's name.
 - symbol: The player's symbol (e.g., 'X' or 'O').
 - **Key Methods:**
 - **Four_in_a_row_player(string name, T symbol):**
 - Constructor that initializes the player's name and symbol.
 - **getmove(int& x, int& y):**
 - Prompts the player to enter the column number where they want to drop their symbol.
 - Validates that the input is within bounds (0 to 6).
 - Finds the correct row by checking from the bottom of the column upwards and places the symbol in the first available space.
-

3. Four_in_a_row_random_player<T>

- **Purpose:**
A computer-controlled player that makes random moves by selecting a valid column and dropping its symbol into the lowest available row in that column.
- **Attributes:**
 - Inherits from RandomPlayer<T>:
 - name: The random player's name is set to "Random Computer Player".
 - symbol: The random player's symbol.
- **Key Methods:**
 - **Four_in_a_row_random_player(T symbol):**
 - Constructor that initializes the random player with a symbol and seeds the random number generator.
 - **getmove(int& x, int& y):**
 - Randomly selects a column y between 0 and 6.
 - Determines the lowest available row x in the selected column and places the symbol in that row.
 - Ensures that the move is valid by checking if the column is full.

• 5x5 Tic Tac Toe

1. X5_O5_Board Class

This class represents the **5x5 game board**. It extends a generic Board<T> base class and implements functionality for managing the board, updating its state, and checking game conditions.

Key Features:

- **Board Initialization:**
 - Initializes a 5x5 grid, setting all cells to 0 (empty).
 - **Game State Functions:**
 - update_board: Updates a cell in the board based on player input.
 - display_board: Displays the board with grid coordinates and player marks.
 - is_win: Checks if a player has won the game (based on a score comparison).
 - is_draw: Checks if the game is a draw (board is full or nearly full with no winner).
 - game_is_over: Combines win/draw checks to determine if the game has ended.
 - **Scoring Functions:**
 - x_score, y_score: Count the number of 3-in-a-row sequences for X and O across rows, columns, and diagonals.
-

2. X5_O5_Player Class

This class defines a **human player** for the 5x5 Tic-Tac-Toe game. It extends a generic Player<T> base class.

Key Features:

- **Constructor:**
 - Takes a name and symbol (X or O) for the player.
 - **Move Input:**
 - Prompts the user to enter their move (row and column) on the board.
-

3. X5_O5_Random_Player Class

This class defines an **AI player that makes random moves** on the board. It extends a RandomPlayer<T> base class.

Key Features:

- **Constructor:**
 - Takes a name and symbol, initializes randomization with a fixed dimension of 5.
 - **Move Logic:**
 - Generates random x and y coordinates (within the 5x5 grid) to place the AI's mark.
-

• Word Tic Tac Toe

1. WordTicTacToe<T>

- **Purpose:**

Represents a word-based Tic-Tac-Toe game where players place letters on the board, and the

goal is to form valid 3-letter words from a provided dictionary. The class handles the game logic, including board layout, checking for valid words, updating moves, and determining win/draw conditions.

- **Attributes:**

- **board:** A 2D array (or pointer to a 2D array) of type T that represents the state of the board, where each element holds the letter placed by a player or is empty (represented as -).
- **valid_words:** A set of valid 3-letter words, read from a provided dictionary file. These words are used to check for a win condition.

- **Key Methods:**

- **WordTicTacToe(int rows, int cols, const string& dict_file):**
Constructor that initializes the game board with the given number of rows and columns, and loads valid 3-letter words from the provided dictionary file.
- **generate_word(int start_x, int start_y, int dx, int dy):**
Generates a 3-letter word starting from a given cell (x, y) and moving in a specified direction (dx, dy). The direction is specified as a change in x and y coordinates (e.g., horizontal, vertical, diagonal).
- **check_for_word(int x, int y):**
Checks if a valid 3-letter word is formed starting from the cell (x, y), in any of the 8 possible directions (horizontal, vertical, diagonal), and either forwards or backwards.
- **update_board(int x, int y, T symbol):**
Updates the board with the player's symbol at the given position (x, y) if the position is valid and unoccupied. Returns true if the move is successful, otherwise false.
- **display_board():**
Displays the current state of the board, with empty cells represented by a dash (-) and filled cells displaying the corresponding symbol.
- **is_win():**
Checks if there is a winner by verifying if any valid 3-letter word has been formed on the board (either horizontally, vertically, or diagonally).
- **is_draw():**
Returns true if the board is full and there is no winner (a draw condition).
- **game_is_over():**
Checks if the game is over, which happens either when there is a winner or when the game reaches a draw.

2. HumanPlayer<T>

- **Purpose:**

Represents a human player in Word Tic-Tac-Toe. It allows the player to interact with the game by providing their move (row, column, and letter) through user input.

- **Attributes:**

- Inherits from the **Player<T>** class, which includes:
 - **name:** The player's name.
 - **symbol:** The player's mark (e.g., 'X' or 'O').

- **Key Methods:**

- **HumanPlayer(const string& player_name):**
Constructor that initializes the player's name and sets the symbol to - (default). The symbol can be updated when the player selects a letter to place on the board.

- **getmove(int& x, int& y):**
Prompts the player to enter their move (row and column). It then asks the player to select a letter to place on the board (converts the letter to uppercase if necessary). The method validates that the input corresponds to a valid cell on the board and a valid letter for the move.

3. ConcreteRandomPlayer<T>

- **Purpose:**
Represents a random player in Word Tic-Tac-Toe. This player automatically chooses a random empty cell on the board to place their symbol without any strategy, making random moves until the game ends.
- **Attributes:**
 - Inherits from the **RandomPlayer<T>** class, which includes:
 - **symbol:** The player's mark (e.g., 'X' or 'O').
 - **dimension:** The size of the game board (rows and columns), used to generate valid random positions for the move.
- **Key Methods:**
 - **ConcreteRandomPlayer(T symbol, int dim):**
Constructor that initializes the random player with the given symbol and the dimension (size) of the board. It also seeds the random number generator to ensure different random moves each time the game is played.
 - **getmove(int& x, int& y):**
Generates a random valid position (x, y) on the board where the player can place their symbol. The method ensures that the selected position is within the bounds of the board and is currently empty. It then outputs the chosen move to the console.

• Numerical Tic Tac Toe

1. Numerical_Tic_Tac_Toe_Board<T>

- **Purpose:**
Represents the game board for Numerical Tic-Tac-Toe, which is a 3x3 grid. This class manages the board layout, handles player moves, and checks for win/draw conditions.
- **Attributes:**
 - **rows:** The number of rows (3 for Numerical Tic-Tac-Toe).
 - **columns:** The number of columns (3 for Numerical Tic-Tac-Toe).
 - **board:** A 2D array (pointer-based) to store the current state of each cell on the board.
 - **n_moves:** The number of moves made so far on the board.
- **Key Methods:**
 - **Numerical_Tic_Tac_Toe_Board():**
Constructor that initializes the board to 3 rows and 3 columns with all cells set to 0 (empty).
 - **~Numerical_Tic_Tac_Toe_Board():**
Destructor that deallocates the memory used by the dynamically allocated board.
 - **display_board():**
Displays the current state of the board, including row and column labels, showing either the number in the cell or leaving it empty if it's 0.

- **update_board(int x, int y, T mark):**
Updates the board at the specified (x, y) position with the given mark (player's symbol). If the mark is 0, it resets the cell (used for undo functionality).
 - **is_win():**
Checks for a winning condition:
 - Checks horizontal, vertical, and diagonal lines to see if a player has aligned 3 numbers whose sum is 15.
 - **is_draw():**
Returns true if the board is full (9 moves) and no winner has been found.
 - **game_is_over():**
Returns true if the game has ended either due to a win or a draw.
-

2. Numerical_Tic_Tac_Toe_player<T>

- **Purpose:**
Represents a human player in Numerical Tic-Tac-Toe. The player selects a number from their available set and places it in the selected position on the board.
 - **Attributes:**
 - **name:** The player's name.
 - **symbol:** The player's symbol (e.g., 1, 3, 5, etc. for odd players and 2, 4, 6, etc. for even players).
 - **available_numbers:** A list of numbers the player can use, depending on whether they are playing with odd or even numbers.
 - **Key Methods:**
 - **Numerical_Tic_Tac_Toe_player(string name, T symbol):**
Constructor that initializes the player's name and symbol, and assigns available numbers (odd or even) based on the player.
 - **getmove(int& x, int& y):**
Prompts the player to enter the row and column coordinates where they want to place their symbol. Validates the coordinates and ensures the player chooses a valid number from their available set.
-

3. Numerical_Tic_Tac_Toe_random_player<T>

- **Purpose:**
A computer-controlled player that makes random moves by selecting a valid number and placing it in a random available position on the board.
- **Attributes:**
 - **name:** The random player's name is set to "Random Computer Player".
 - **symbol:** The random player's symbol (e.g., 1, 3, 5, etc. for odd players and 2, 4, 6, etc. for even players).
 - **available_numbers:** A list of numbers the random player can use, alternating between odd and even numbers.
- **Key Methods:**
 - **Numerical_Tic_Tac_Toe_random_player(T symbol):**
Constructor that initializes the random player with a symbol and seeds the random number generator for making random moves.

- **getmove(int& x, int& y):**
Randomly selects a row and column (x and y) and randomly selects a number from the player's available set. The selected number is then removed from the available set
-

• Misere Tic Tac Toe

1. X6_O6_Board Class

This class represents the **6x6 game board**. It extends a generic Board<T> base class and includes methods for updating, displaying, and checking game conditions.

Key Features:

- **Board Initialization:**
 - A 6x6 grid is initialized with all cells set to 0 (empty).
 - **Game State Functions:**
 - update_board: Allows updates to the board if the move is valid (checks for boundary and already occupied cells).
 - display_board: Displays the current board state with each cell showing its coordinates and value.
 - is_win: Checks if there is a winning condition (3 in a row, column, or diagonal).
 - is_draw: Determines if the game is a draw (board is full and no winner).
 - game_is_over: Combines the is_win and is_draw methods to return whether the game has concluded.
-

2. X6_O6_Player Class

This class defines a **human player** for the 6x6 Tic-Tac-Toe game, inheriting from the generic Player<T> class.

Key Features:

- **Constructor:**
 - Initializes the player's name and symbol (X or O).
 - **Move Input:**
 - Prompts the human player to input their move by specifying x and y coordinates (from 0 to 4).
-

3. X6_O6_Random_Player Class

This class defines an **AI player that randomly makes moves**. It inherits from the RandomPlayer<T> class.

Key Features:

- **Constructor:**
 - Takes a name and symbol for initialization, then sets the dimension to 5.
 - Initializes random seed based on the current system time.
 - **Move Logic:**
 - Uses rand() to select random x and y coordinates within the 5x5 grid to make a move.
-

• Report on Code Quality and Issue Resolution:

As part of our collaborative project, reviewing and improving each other's code played a pivotal role in ensuring a polished final product. During this process, we identified several challenges and mistakes in the code contributed by team members, which we systematically resolved. Here are the key issues we encountered and how they were addressed:

1. Input Handling Issues

In the *Pyramid Tic-Tac-Toe* game, there were critical gaps in input validation. Specifically:

- **Out-of-Range Input:** No mechanism was in place to handle moves that exceeded the pyramid's dimensions.
- **Random Player Input Validation:** The random player also lacked safeguards against generating invalid moves.

To resolve these issues, we implemented a robust input validation loop that checks for valid integer input within the allowed range. This ensures the game runs smoothly and prevents unexpected crashes. The following code snippet demonstrates our solution:

```
while (true) {  
    cout << "\nPlease enter your move x and y (0 to 2) separated by spaces:";  
    cin >> x >> y;  
  
    // Check if input is valid  
    if (cin.fail() || x < 0 || x > 2 || y < 0 || y > 2) {  
        cin.clear(); // Clear the fail state  
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Discard invalid input  
        cout << "\nInvalid input. Please enter valid coordinates (0, 1, or 2).\n";  
    } else {  
        break; // Exit loop if input is valid  
    }  
}
```

This code addition ensures the game handles invalid inputs gracefully and avoids infinite output loops, such as when non-numeric input (e.g., a character) is entered.

2. Algorithm Challenges in Ultimate Tic-Tac-Toe (9x9)

While designing the *Ultimate Tic-Tac-Toe* game with a 9x9 board, we initially attempted to implement it using an array of pointers to standard Tic-Tac-Toe boards. However, this approach encountered pointer-related issues that made the solution overly complex and error-prone.

To overcome this, we pivoted to a more efficient approach:

- We implemented the 9x9 board as a regular 2D array.
- Added boolean flags to track sub-board states and transitions.

This method simplified the implementation and reduced the risk of bugs, while maintaining the flexibility required for the larger game board.

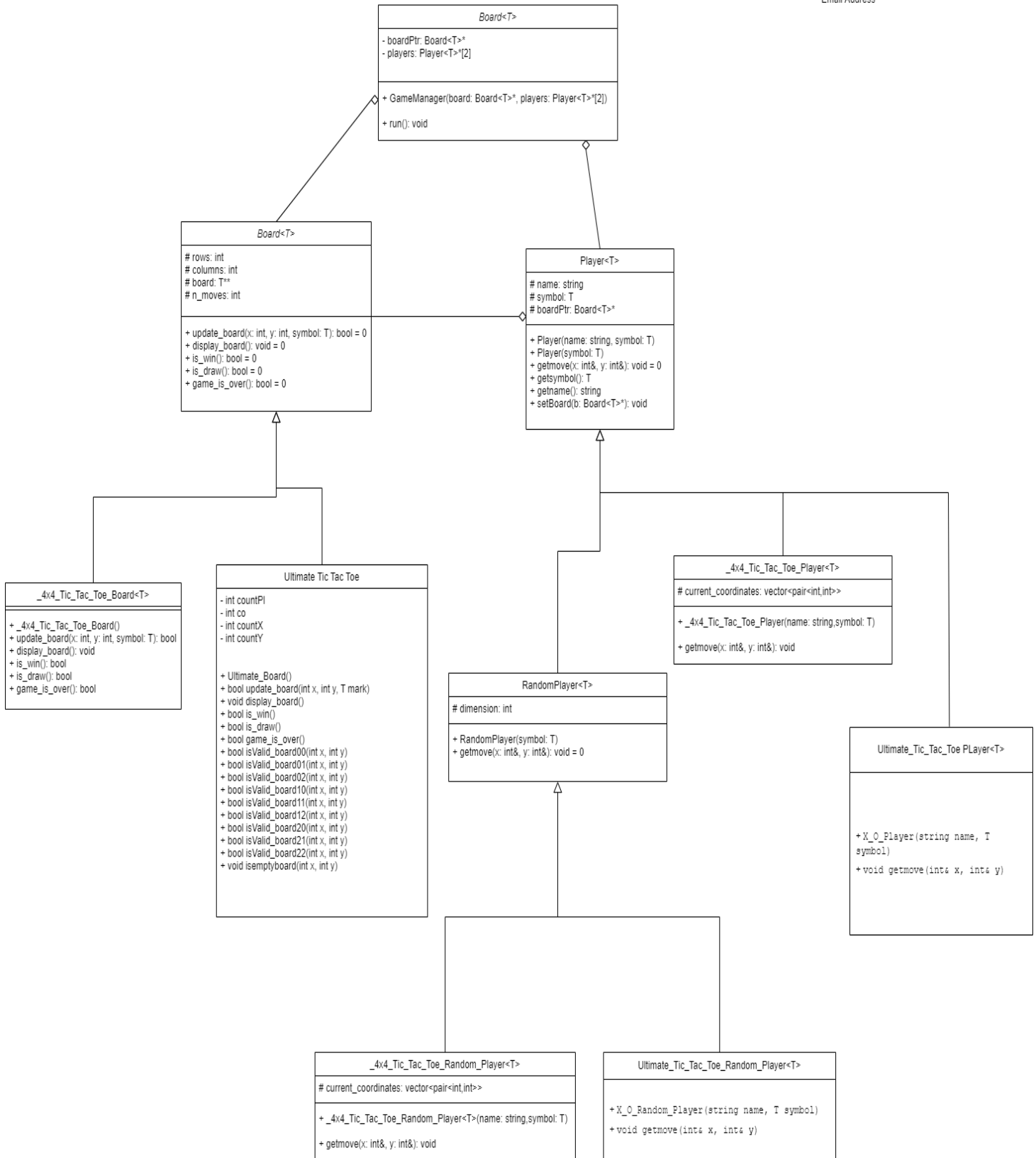
3. General Observations on Code Quality

- **Error Handling:** The original codebase lacked sufficient error-handling mechanisms, especially for user input. We emphasized improving this aspect across all games.
- **Code Readability:** Some team members' code sections were written with minimal comments, making it harder to debug and understand. We incorporated descriptive comments and consistent formatting for better maintainability.
- **Bug Resolution:** Detecting and fixing bugs collaboratively enhanced the overall stability of the project. This included addressing infinite loops and crashes caused by invalid data types or out-of-range values.


By focusing on these improvements, we ensured that the final codebase adheres to best practices, is user-friendly, and operates without major errors. This process also emphasized the importance of teamwork and iterative problem-solving in software development.


• Bounce games' UML class diagram.


Email Address





• Git Hub repository screen shot:


 **Assignment-2-Games** Public


 Watch **1**

 Fork **0**

 Star **0**

 master

 2 Branches

 0 Tags

Q Go to file


Add file

<> Code

About


This branch is **11 commits ahead of**, **2 commits behind** `main`.


Contribute

 **Eng-WalidAdel** Merge remote-tracking branch 'origin/master' 4ff873a · 13 hours ago 11 Commits


.idea	Merge remote-tracking branch 'origin/master'	13 hours ago
Assignment demo without AI Bonus	some changes	13 hours ago
Mustafa's_code	Mustafa's game code.	yesterday
cmake-build-debug	some changes	13 hours ago
3x3X_O.h	some changes	13 hours ago
5x5X_O.h	some changes	13 hours ago
BoardGame_Classes.h	some changes	13 hours ago
CMakeLists.txt	some changes	13 hours ago
Game6.h	some changes	13 hours ago
implementation.h	starting point guys	3 weeks ago
main.cpp	some changes	13 hours ago


README

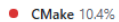
 **Eng-M0stafaEhab** Mostafa Ehab Mosta...


 **Eng-WalidAdel** ENG- Walid Adel

Languages

 C++ 61.1%

 C 25.0%

 CMake 10.4%

 HTML 3.5%

- **Team members' work break-down:**

Before getting into individual responsibilities, it's essential to note that this project was carried out in a highly collaborative environment. Each team member contributed actively to their tasks while also supporting each other to ensure the project's success. This mutual assistance and collective effort made a significant impact on the quality and completion of the project.

Members' names	Tasks
Sara Ibrahim Mohamed	<ul style="list-style-type: none">• Game 1• Game 4
Walid Adel Mordy Rohyem	<ul style="list-style-type: none">• Game 3• Game 6• Bonus: Game 8
Mustafa Ehab Mustafa Akl	<ul style="list-style-type: none">• Game 2• Game 5• Bonus: Game 7• Report

- **To take a look at the GitHub repository and the code separate files [click here](#) (Ctrl + left click)(All the content in the master branch).**