Faculty of Computers and Artificial Intelligence

Cairo University

CS213: OOP

Dr. Mohamed El-Ramly

Assignment 2 Task2,3,4,5

Team Members and IDs:

- Mohamed Ahmed Mohamed 20230316
- Mohamed Mahmoud Bayoumi 20230620
- Moamen Desouki Mahmoud 20230431

Table of Contents

1.Who made what:

Name	Work Done
Mohamed ElGabry	**Game 3,6,7 / Menu / Report**
Mohamed ElTanahy	**Game 1,4,8 / Ultimate Game GUI**
Moamen Hamam	**Game 2,5 / Four by Four Game GUI**

2. Games Descriptions:

2.1. Pyramid Tic-Tac-Toe Game Classes

The Pyramid X-O game is a variant of Tic-Tac-Toe played on a pyramid-shaped board. This board has a triangular layout with progressively increasing columns in each row, starting from one column in the top row to five columns in the bottom row. Players aim to form winning patterns (typically rows, columns, or diagonals).

PyramidX_O_Board<T>

This class represents the triangular board for the Pyramid X-O game.

- Constructor:
    - Initializes a 3-row triangular board with 1 column in the first row, 3 in the second, and 5 in the third.
    - Fills the board with zeros (empty state).
    - Keeps track of the number of moves made.
- Key Methods:
    - update_board(int x, int y, T symbol): Updates the board at a specific position if the move is valid.
        - Allows clearing a cell with symbol=0.
        - Ensures placement within the triangular structure.
    - display_board(): Displays the pyramid board with proper alignment for the triangular layout.
    - is_win(): Checks for winning conditions in rows, columns, or diagonals.
    - is_draw(): Returns true if all moves are completed and no winner exists.
    - game_is_over(): Combines is_win and is_draw to determine if the game has ended.

PyramidX_0_Player<T>

This class defines a human player in the Pyramid X-O game.

- Constructor:
    - Initializes a player with a name and a game symbol (X or O).
- Key Method:
    - getmove(int &x, int &y): Prompts the player to input their move coordinates.

PyramidX_0_Random_Player<T>

This class represents a random computer player.

- Constructor:
    - Initializes the player with a symbol and sets the player's name as "Random Computer Player."
    - Seeds the random number generator.
- Key Method:
    - getmove(int &x, int &y): Generates random coordinates within the valid triangular range.

2.2. Four in a Row Classes

Four in a Row (Connect 4) is a classic game where players drop tokens into a 7-column by 6-row board, aiming to align four tokens vertically, horizontally, or diagonally.

FourInARowBoard<T>

This class represents the Connect 4 board.

- Constructor:
    - Initializes a 6x7 grid with empty cells (0).
    - Keeps track of the number of moves made.
- Key Methods:
    - update_board(int x, int y, T symbol): Places a symbol in the lowest available cell of a given column.
    - display_board(): Displays the grid-style board with separators.
    - is_win(): Combines horizontal, vertical, and diagonal checks to determine a win.
    - is_draw(): Checks if all 42 cells are filled without a winner.

- game_is_over(): Determines if the game has concluded.
- Internal helper methods (check_horizontal, check_vertical, check_diagonal): Detect winning alignments in respective directions.

FourInARowPlayer<T>

This class defines a human player for Four in a Row.

- Constructor:
  - Initializes the player with a name and symbol.
- Key Methods:
  - getmove(int &x, int &y, int &number): Prompts the player to choose a column to drop their token.

FourInARow_Random_Player<T>

This class represents a random computer player.

- Constructor:
  - Initializes the player with a symbol and sets their name to "Random Computer Player."
- Key Methods:
  - getmove(int &x, int &y, int &number): Randomly selects a column to drop the token.

2.3. Five-by-Five Tic-Tac-Toe Game Classes

This is a version of Tic-Tac-Toe played on a 5x5 grid. Players aim to form patterns such as rows, columns, or diagonals of three consecutive symbols (X or O). By the end of the game when they reach 24 moves, The winner gets decided by the player with more score.

FiveX_O_Board<T>

This class represents the 5x5 game board.

- Constructor:
  - Initializes a 5x5 grid with empty cells (0).
  - Tracks the number of moves made.
  - Initializes counters for counting patterns (Xcounter and Ocounter).
- Key Methods:
  - update_board(int x, int y, T symbol): Updates a cell with a symbol if valid.

- display_board(): Prints the 5x5 board with proper alignment.
- is_win(): Checks for winning conditions (row, column, diagonal).
- is_draw(): Returns true if all moves are completed without a winner.
- game_is_over(): Combines is_win and is_draw.
- count_patterns(): Identifies patterns of three consecutive symbols in rows, columns, and diagonals.
- reset_counters(): Resets pattern counters to zero.

FiveX_O_Player<T>

This class defines a human player in the 5x5 X-O game.

- Constructor:
  - Initializes the player with a name and a symbol.
- Key Method:
  - getmove(int &x, int &y): Prompts the player to input their move coordinates.

FiveX_O_Random_Player<T>

This class represents a random computer player.

- Constructor:
  - Initializes the player with a symbol and sets their name to "Random Computer Player."
- Key Methods:
  - getmove(int &x, int &y): Randomly selects a valid cell for placing a symbol.

2.4. Word Tic-Tac-Toe Game Classes

The objective is to form a valid word (from a dictionary) by placing X's and O's in a row, column, or diagonal. Players must strategically place their marks to either create a word or block their opponent from doing so.[1]

WordX_O_Board<T>

- Purpose: Manages the board for the WordX_O game.

- Key Members:
    - dictionary: An unordered map that stores words from a dictionary loaded from a file. These words are checked as possible winning conditions.
    - Constructor: The constructor opens a file containing words, loads them into the dictionary, and initializes a 3x3 game board with all cells set to 0 (empty).
    - update_board(x, y, mark): Updates the board with a given symbol ('X' or 'O') at the specified coordinates. It also allows for undoing moves.
    - display_board(): Displays the board with the current symbols and their positions.
    - is_win(): Checks if there is a winning condition on any row, column, or diagonal, by checking if any combination of three consecutive symbols forms a valid word in the dictionary.
    - is_draw(): Returns true if there are 9 moves and no winner.
    - game_is_over(): Returns true if the game is over (either by win or draw).

WordX_O_Player<T>

- Purpose: Represents a player in the WordX_O game.
- Key Members:
    - Constructor: Takes the player's name and symbol (either 'X' or 'O') and initializes the player.
    - getmove(x, y): Prompts the player to enter their move coordinates (x, y), followed by the symbol they want to place on the board.

WordX_O_Random_Player<T>

- Purpose: Represents a random computer player in the WordX_O game.
- Key Members:
    - Constructor: Initializes a random computer player with a given symbol.
    - getmove(x, y): Generates a random move for the player (randomly selects x, y coordinates and a random symbol).

2.5. Numerical Tic-Tac-Toe Game Classes

The objective is to place numbers (1-9) in a row, column, or diagonal where the sum of the three numbers equals 15. Players must choose numbers wisely to reach the sum of 15 before their opponent does.

NumX_O_Board<T>

- Purpose: Manages the board for the NumericalX_O game.
- Key Members:
    - Constructor: Initializes a 3x3 board with zeroes, representing empty spaces.
    - update_board(x, y, number): Places the specified number on the board at the given coordinates and updates the number of moves.
    - display_board(): Displays the current state of the board with the numbers placed by players.
    - check_sum(a, b, c): Checks if the sum of three numbers (a, b, c) is equal to 15. This is the winning condition in NumericalX_O.
    - is_win(): Checks if any row, column, or diagonal has a sum equal to 15, indicating a win.
    - is_draw(): Returns true if all 9 moves have been made and there is no winner (sum is not 15).
    - game_is_over(): Returns true if the game has ended due to a win or a draw.

NumX_O_Player<T>

- Purpose: Represents a player in the NumericalX_O game.
- Key Members:
    - Constructor: Initializes the player's name, symbol, and whether they play with odd or even numbers.
    - getmove(x, y, number): Prompts the player to enter the row, column, and number they wish to place on the board.
    - isNumberChosen(number): Checks if a given number has already been chosen by the player.

NumX_O_Random_Player<T>

- Purpose: Represents a random computer player in the NumericalX_O game.
- Key Members:
    - Constructor: Initializes the random player with a given symbol.
    - getmove(x, y, number): Generates random coordinates (x, y) and a random number between 1 and 9 for the computer player.

2.6. Inverse(Misere) Tic-Tac-Toe Game Classes

The goal is to avoid winning. Players try not to place three identical symbols in a row, column, or diagonal. If a player does so, they lose the game. It's a "Misère" (reverse) version of Tic-Tac-Toe, where making a winning move results in defeat.

InverseX_O_Board Class<T>

Purpose: Manages the board for the InverseX_O (Misere) game.

- Key Members:
    - win: A static boolean flag to indicate whether the game is won.
    - Constructor: Initializes a 3x3 board with empty cells (represented as 0), and sets the number of moves to 0. The win condition is reset at the start of the game.
    - update_board(x, y, mark): Updates the board with the specified symbol ('X' or 'O') at the coordinates x, y. The game checks if the current player has won and updates the board accordingly.
    - display_board(): Displays the current state of the board.
    - is_win(): Checks if there is a winning condition (three identical symbols in a row, column, or diagonal). The win condition is inverted (the player who makes the winning move loses in this version).
    - is_draw(): Returns true if all 9 moves are completed without a winner, indicating a draw.
    - game_is_over(): Returns true if the game has ended, either by win or draw.

InverseX_O_Player<T>

- Purpose: Represents a player in the InverseX_O game.
- Key Members:
    - Constructor: Initializes the player's name and symbol ('X' or 'O').
    - getmove(x, y): Prompts the player to input the coordinates (x, y) for their move.

InverseX_O_Random_Player<T>

Purpose: Represents a random computer player in the InverseX_O game.

- Key Members:
    - Constructor: Initializes a random computer player with the given symbol.
    - getmove(x, y): Generates random coordinates (x, y) for the computer's move.

2.7. Four Tic-Tac-Toe Game Classes

The game starts with 4 Xs and 4 Os on the board which you can move vertically or horizontally till one of the players connect three in a row.

FourX_O_Board<T>

- **Description**: This class extends the Board class to represent a 4x4 game board for the "Four X-O" game. It manages the board's layout, updates, and checks for game-ending conditions like wins or draws.
- **Key Methods**:
    - FourX_O_Board(): Constructor that initializes a 4x4 board with alternating 'X' and 'O' symbols on the top and bottom rows.
    - update_board(int x, int y, T symbol): Updates the board based on the move from one position to another. Validates the move to ensure it adheres to game rules.
    - display_board(): Prints the current state of the board to the console.
    - is_win(): Checks if a player has won by aligning three consecutive symbols in a row, column, or diagonal.
    - is_draw(): Placeholder method; currently always returns false.
    - game_is_over(): Returns true if the game has ended due to a win or draw.

FourX_O_Player<T>

- **Description**: This class extends the Player class and represents a human player in the game. It manages the player's name, symbol, and their moves.
- **Key Methods**:
    - FourX_O_Player(string name, T symbol): Constructor to initialize the player's name and symbol.
    - getmove(int& x, int& y): Prompts the player to input the starting position (from_x, from_y) and the target position (to_x, to_y). Combines these values into a single move representation.

FourX_O_Random_Player<T>

- **Description**: This class extends the RandomPlayer class and represents a computer player that makes random moves on the board.
- **Key Methods**:
    - FourX_O_Random_Player(T symbol): Constructor that initializes the player's symbol and seeds the random number generator.

- getmove(int& x, int& y): Generates random positions for the move (from_x, from_y and to_x, to_y) within the 4x4 grid.

2.8. ULTIMATE Tic-Tac-Toe Game Classes

This game consists of a 3x3 grid of smaller Tic-Tac-Toe boards, where each move made by a player determines the sub-board where their opponent must play next. The objective is to win three sub-boards in a row (horizontally, vertically, or diagonally) on the larger grid.

ULT_X_O_Board<T>

- **Description**: This class extends the Board class to represent the "Ultimate Tic-Tac-Toe" game board. It uses nine 3x3 sub-boards (X_O_Board) to create a 3x3 grid of boards. Players win the game by aligning wins in three sub-boards in a row, column, or diagonal.
- **Key Methods**:
  - ULT_X_O_Board(): Constructor that initializes the 3x3 grid of sub-boards (ult) and the corresponding 3x3 board to track sub-board wins.
  - update_board(int x, int y, T symbol): Handles moves, translating global coordinates (across the ultimate board) into the respective sub-board and local coordinates.
  - display_board(): Prints both the ultimate board with all sub-board states and the simplified win board that tracks sub-board victories.
  - mini_win(): Updates the ultimate board by marking sub-boards as won if a player has achieved a win in that sub-board.
  - is_win(): Checks if a player has won the ultimate game by aligning three sub-boards in a row, column, or diagonal on the win board.
  - is_draw(): Placeholder method; currently always returns false.
  - game_is_over(): Returns true if the game ends due to a win or draw.

ULT_X_O_Player<T>

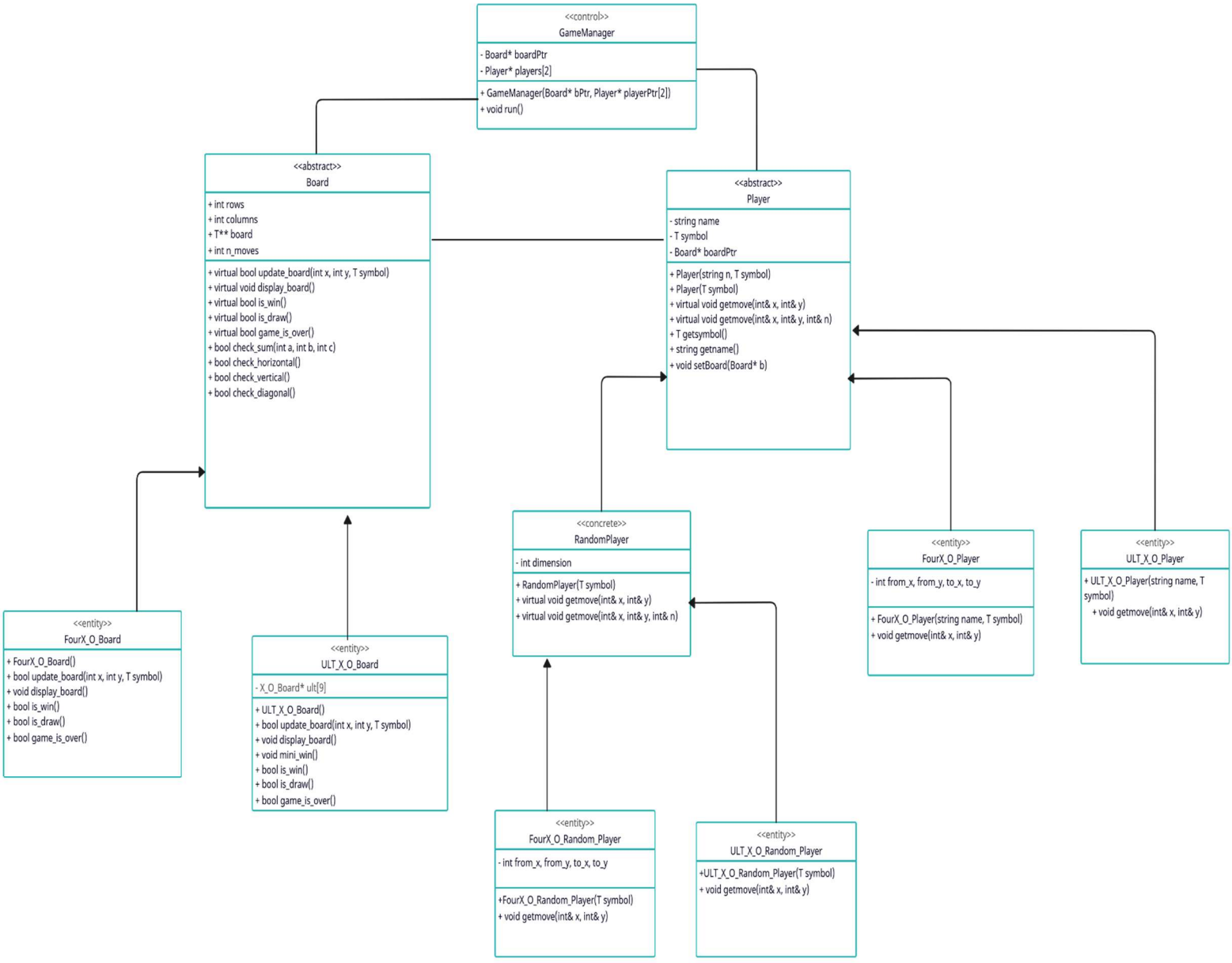
- **Description**: This class extends the Player class and represents a human player in the Ultimate Tic-Tac-Toe game. It manages the player's name, symbol, and their moves.
- **Key Methods**:
  - ULT_X_O_Player(string name, T symbol): Constructor to initialize the player's name and symbol.

- getmove(int& x, int& y): Prompts the player to input global coordinates (x and y) for their move.

ULT_X_O_Random_Player<T>

- **Description**: This class extends the RandomPlayer class and represents a computer player that makes random moves within the 9x9 grid.
- **Key Methods**:
  - ULT_X_O_Random_Player(T symbol): Constructor that initializes the player's symbol and seeds the random number generator.
  - getmove(int& x, int& y): Generates random global coordinates for the move (x and y) within the 9x9 grid.

3. UML Class Diagrams:



4. Individual Quality Reports:

4.1. Mohamed ElGabry's Report:

1. FourInARow Header File

Strengths:

1. **Separation of Concerns**:
    - The header file separates the core **game logic** (like checking win conditions) from other components (like the player class). This improves modularity.
2. **Logical Game Mechanics**:
    - Methods for **placing pieces** (update_board) and **checking win conditions** (check_horizontal, check_vertical, check_diagonal) are well-implemented.

Weaknesses:

1. **Input Validation**:
    - There is limited error handling for invalid user input (e.g., invalid columns).[2]
2. **Empty Methods**:
    - Some methods like getmove(int&, int&) are empty or unused. These add clutter to the file.

Recommendations:

- Add a **destructor** to free dynamically allocated memory.
- Improve input validation to handle invalid or out-of-bound moves gracefully.
- Remove or explicitly mark unused methods for future implementation.

---

2. Pyramid Header File

Strengths:

1 **Template Support**:

- Similar to the "Four in a Row" header, the usage of template<typename T> increases flexibility.

Weaknesses:

1. **Lack of Documentation**:
   - The header file lacks comments or documentation explaining the **pyramid structure** or the rules of the game.
2. **Dynamic Memory Issues**:
   - Like the "Four in a Row" file, dynamic memory allocation is not followed by a **destructor**.
3. **Logic Complexity**:
   - Without comments, the complexity of the pyramid structure and its indexing may confuse readers.

Recommendations:

- Add comments and documentation to describe the pyramid structure and rules.
- Add a destructor for memory cleanup.
- Simplify or document complex logic for pyramid indexing.
- Validate player input to ensure it matches the expected format.

---

3. Numerical Header File

Strengths:

1. **Template Flexibility**:
   - As with the other headers, template<typename T> ensures that the board can support various numerical types.
2. **Logical Flow**:
   - Win-checking conditions and updates to the board appear logically structured.

Weaknesses:

1. **Ambiguity in Game Rules**:
    - Without proper documentation, it is unclear how numbers are used (e.g., addition, subtraction, or comparisons).
2. **Input Validation**:
    - There might be insufficient handling for invalid numerical inputs.
3. **Dynamic Memory Management**:
    - Similar to other files, dynamic memory allocation for the board lacks a destructor.
4. **Complex Logic Without Comments**:
    - If win-checking involves numerical operations, the logic can become complex and unreadable without comments.

Recommendations:

- Add clear documentation explaining the game rules and numerical operations.
- Allow dynamic board dimensions or configurable numeric ranges.
- Add a destructor to manage dynamic memory properly.
- Validate user input to ensure it meets numerical requirements.
- Simplify or document complex logic for numerical win-checking.

---

4. Word Header File

Strengths:

1. **Dictionary Integration**:
    - The game uses **file-based word input** and unordered_map to validate words. This adds sophistication and scalability.
2. **Flexible Gameplay**:
    - Players can interact using word-based moves, which is unique compared to traditional board games.
3. **Logical Structure**:

- Methods like is_win for checking rows, columns, and diagonals follow a consistent pattern.

Weaknesses:

1. **File Input Handling**:
   - The header file assumes valid file input without robust exception handling for missing or incorrect files.
2. **Redundant Logic**:
   - Checking rows, columns, and diagonals for valid words may contain redundant code that can be refactored into a helper function.
3. **Dynamic Memory Issues**:
   - As with other headers, dynamic memory allocation for the board lacks a destructor.
4. **Input Validation**:
   - There's no robust validation to ensure players input only single letters or valid moves.

Recommendations:

- Add exception handling for word file input failures.
- Allow dynamic board size using constructor parameters.
- Refactor win-checking logic into a reusable helper function.
- Add a destructor to handle dynamic memory cleanup.
- Strengthen input validation to ensure players enter valid moves.

4.2. Mohamed ElTanahy's Report:

1. Inverse Header File:

Strengths:

Well-Defined Class Structure

- Succefully used thegameboard class to inherit and make the code.

Readable and Logical Implementation

- The game mechanics are thoughtfully implemented with clear functions like update_board, is_win, and is_draw. These methods are easy to understand and provide a solid foundation for gameplay logic.
- Clear variable names (e.g., n_moves, mark) and well-placed comments enhance the readability of the code.

Random Player Integration

- Works as intended

Game Functionality

- The board update and display mechanisms work together seamlessly, providing real-time feedback to players. The logic for determining win conditions, draws, and game-over states is well-structured.

Defensive programming

- Basic safeguards are in place to handle invalid moves and ensure the board's integrity. This is crucial for a smooth gameplay experience.

Weaknesses:

1. **Win Detection Logic**
   - **Observation**: The is_win method uses repetitive logic for checking rows, columns, and diagonals.
2. **Static Win State**
   - **Concern**: The static bool win variable in the board class introduces potential issues with multiple game instances.

2.FivexFive Header File:

Strengths:

1. **Scalable Gameplay**
   - Expanding the board size to 5x5 and incorporating patterns to track scores adds an interesting twist to the classic game. This change enhances the game's complexity and replayability.

2. **Pattern Recognition Logic**
    - The count_patterns method is a notable addition that effectively tracks scoring based on patterns. This feature adds depth to the gameplay by evaluating multiple winning conditions.
3. **Incremental Updates to Gameplay Rules**
    - Modifications to is_win and is_draw reflect the extended gameplay, ensuring that the game logic adapts to the larger board and scoring system.
4. **Comprehensive Board Display**
    - The display_board function presents the board state with clear row-column mapping, helping players visualize their moves effectively.

Weaknesses:

1. **Code Duplication in count_patterns**

- **Observation:** The logic for detecting patterns (rows, columns, and diagonals) is repeated with minor variations.

2. **Score Display in is_win**

- **Observation:** The is_win method mixes game logic with user-facing output (cout).

3 **Limited Error Handling for Player Input**

- **Observation:** The getmove function assumes valid input from users but lacks validation for out-of-range or invalid moves.
- **Recommendation:** Add checks for valid input and re-prompt the user if needed.

4. **Static n_moves in Players**

- **Risk:** The n_moves counter in both FiveX_O_Player and FiveX_O_Random_Player is initialized but not effectively tied to the board's state.
- **Recommendation:** Use FiveX_O_Board::n_moves directly to maintain consistency across components.

---

3. Numerical Header File

Strengths:

3. **Template Flexibility**:

- As with the other headers, template<typename T> ensures that the board can support various numerical types.

4. **Logical Flow**:
    - Win-checking conditions and updates to the board appear logically structured.

Weaknesses:

5. **Ambiguity in Game Rules**:
    - Without proper documentation, it is unclear how numbers are used (e.g., addition, subtraction, or comparisons).
6. **Input Validation**:
    - There might be insufficient handling for invalid numerical inputs.
7. **Dynamic Memory Management**:
    - Similar to other files, dynamic memory allocation for the board lacks a destructor.
8. **Complex Logic Without Comments**:
    - If win-checking involves numerical operations, the logic can become complex and unreadable without comments.

Recommendations:

- Add clear documentation explaining the game rules and numerical operations.
- Allow dynamic board dimensions or configurable numeric ranges.
- Add a destructor to manage dynamic memory properly.
- Validate user input to ensure it meets numerical requirements.
- Simplify or document complex logic for numerical win-checking.

---

4. Word Header File

Strengths:

4. **Dictionary Integration**:
    - The game uses **file-based word input** and unordered_map to validate words. This adds sophistication and scalability.
5. **Flexible Gameplay**:

- Players can interact using word-based moves, which is unique compared to traditional board games.

6. **Logical Structure**:
    - Methods like is_win for checking rows, columns, and diagonals follow a consistent pattern.

Weaknesses:

5. **File Input Handling**:
    - The header file assumes valid file input without robust exception handling for missing or incorrect files.
6. **Redundant Logic**:
    - Checking rows, columns, and diagonals for valid words may contain redundant code that can be refactored into a helper function.
7. **Dynamic Memory Issues**:
    - As with other headers, dynamic memory allocation for the board lacks a destructor.
8. **Input Validation**:
    - There's no robust validation to ensure players input only single letters or valid moves.

Recommendations:

- Add exception handling for word file input failures.
- Allow dynamic board size using constructor parameters.
- Refactor win-checking logic into a reusable helper function.
- Add a destructor to handle dynamic memory cleanup.
- Strengthen input validation to ensure players enter valid moves.

4.3. Moamen ElDesouki's Report:

1. FourInARow Header File

Strengths:

3. **Separation of Concerns**:
    - The header file separates the core **game logic** (like checking win conditions) from other components (like the player class). This improves modularity.
4. **Logical Game Mechanics**:
    - Methods for **placing pieces** (update_board) and **checking win conditions** (check_horizontal, check_vertical, check_diagonal) are well-implemented.

Weaknesses:

3. **Input Validation**:
    - There is limited error handling for invalid user input (e.g., invalid columns).[3]
4. **Empty Methods**:
    - Some methods like getmove(int&, int&) are empty or unused. These add clutter to the file.

Recommendations:

- Add a **destructor** to free dynamically allocated memory.
- Improve input validation to handle invalid or out-of-bound moves gracefully.
- Remove or explicitly mark unused methods for future implementation.

---

2. Pyramid Header File

Strengths:

1. **Template Support**:
    - Similar to the "Four in a Row" header, the usage of template<typename T> increases flexibility.

Weaknesses:

4. **Lack of Documentation**:
    - The header file lacks comments or documentation explaining the **pyramid structure** or the rules of the game.
5. **Dynamic Memory Issues**:
    - Like the "Four in a Row" file, dynamic memory allocation is not followed by a **destructor**.
6. **Logic Complexity**:
    - Without comments, the complexity of the pyramid structure and its indexing may confuse readers.

Recommendations:

- Add comments and documentation to describe the pyramid structure and rules.
- Add a destructor for memory cleanup.
- Simplify or document complex logic for pyramid indexing.
- Validate player input to ensure it matches the expected format.

---

3. **Inverse Header File:**

Strengths:

Well-Defined Class Structure

- Succefully used thegameboard class to inherit and make the code.

Readable and Logical Implementation

- The game mechanics are thoughtfully implemented with clear functions like update_board, is_win, and is_draw. These methods are easy to understand and provide a solid foundation for gameplay logic.
- Clear variable names (e.g., n_moves, mark) and well-placed comments enhance the readability of the code.

Random Player Integration

- Works as intended

Game Functionality

- The board update and display mechanisms work together seamlessly, providing real-time feedback to players. The logic for determining win conditions, draws, and game-over states is well-structured.

Defensive programming

- Basic safeguards are in place to handle invalid moves and ensure the board's integrity. This is crucial for a smooth gameplay experience.

Weaknesses:

1. **Win Detection Logic**
   - **Observation**: The is_win method uses repetitive logic for checking rows, columns, and diagonals.
2. **Static Win State**
   - **Concern**: The static bool win variable in the board class introduces potential issues with multiple game instances.

4.FivexFive Header File:

Strengths:

1. **Scalable Gameplay**
   - Expanding the board size to 5x5 and incorporating patterns to track scores adds an interesting twist to the classic game. This change enhances the game's complexity and replayability.

2. **Pattern Recognition Logic**
    - The count_patterns method is a notable addition that effectively tracks scoring based on patterns. This feature adds depth to the gameplay by evaluating multiple winning conditions.
3. **Incremental Updates to Gameplay Rules**
    - Modifications to is_win and is_draw reflect the extended gameplay, ensuring that the game logic adapts to the larger board and scoring system.
4. **Comprehensive Board Display**
    - The display_board function presents the board state with clear row-column mapping, helping players visualize their moves effectively.

Weaknesses:

1. **Code Duplication in count_patterns**

- **Observation:** The logic for detecting patterns (rows, columns, and diagonals) is repeated with minor variations.

2. **Score Display in is_win**

- **Observation:** The is_win method mixes game logic with user-facing output (cout).

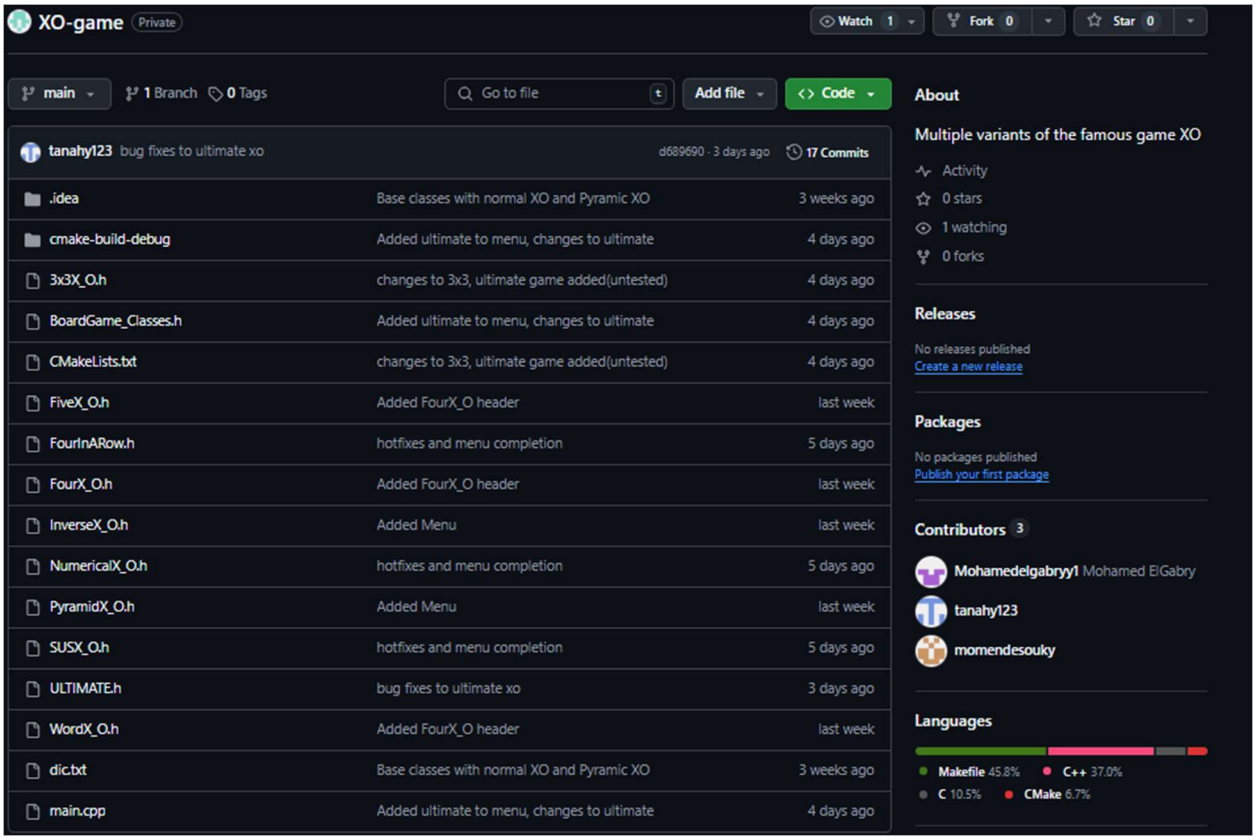
3 **Limited Error Handling for Player Input**

- **Observation:** The getmove function assumes valid input from users but lacks validation for out-of-range or invalid moves.
- **Recommendation:** Add checks for valid input and re-prompt the user if needed.

4. **Static n_moves in Players**

- **Risk:** The n_moves counter in both FiveX_O_Player and FiveX_O_Random_Player is initialized but not effectively tied to the board's state.
- **Recommendation:** Use FiveX_O_Board::n_moves directly to maintain consistency across components.

5. Github:



Link: https://github.com/Tanahy05/XO-game

6. GUI DEMO video link:

https://youtu.be/ytHLNkPOco8