Faculty of Computers and Artificial Intelligence
Cairo University
CS213: Object Oriented Programming
Dr. Mohammad El-Ramly

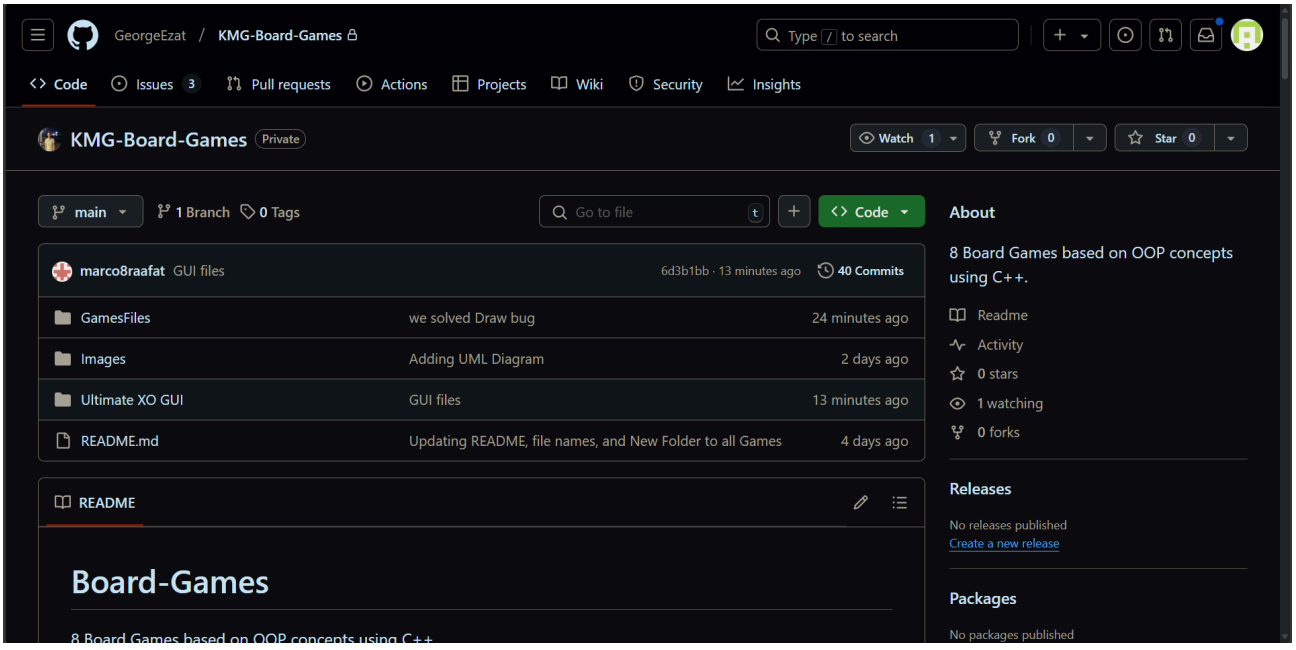
Assignment_2_T2,3,4,5

Name	Email	Group	ID
George Ezat Hosni	georgeezat248@gmail.com	B_S17	20231041
Keorlus Akram Mady	Kerolus.Akram@gmail.com	B_S17	20231126
Marco Raafat Zakaria	marco782005@gmail.com	B_S17	20231129

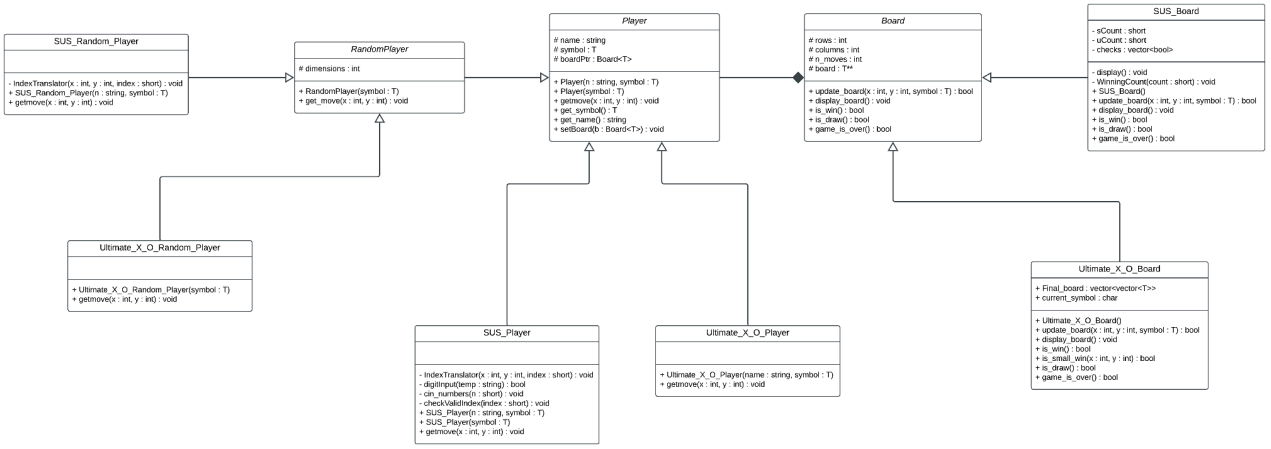
Responsibilities:

George Ezat Hosni	• GitHub Administration • Files Made: Pyramid-Tic-Tac-Toe.h **WITH AI** Word-Tic-Tac-Toe.h SUS.h KMG-Manager KMG-Board-Games.cpp • UML Diagram
Kerolus Akram Mady	• Files Made: Four-in-a-row.h Numerical-Tic-Tac-Toe.h Ultimate-Tic-Tac-Toe.h **WITH GUI**
Marco Raafat Zakaria	• Files Made: 5x5-Tic-Tac-Toe.h Misere-Tic-Tac-Toe.h Ultimate-Tic-Tac-Toe.h **WITH GUI**

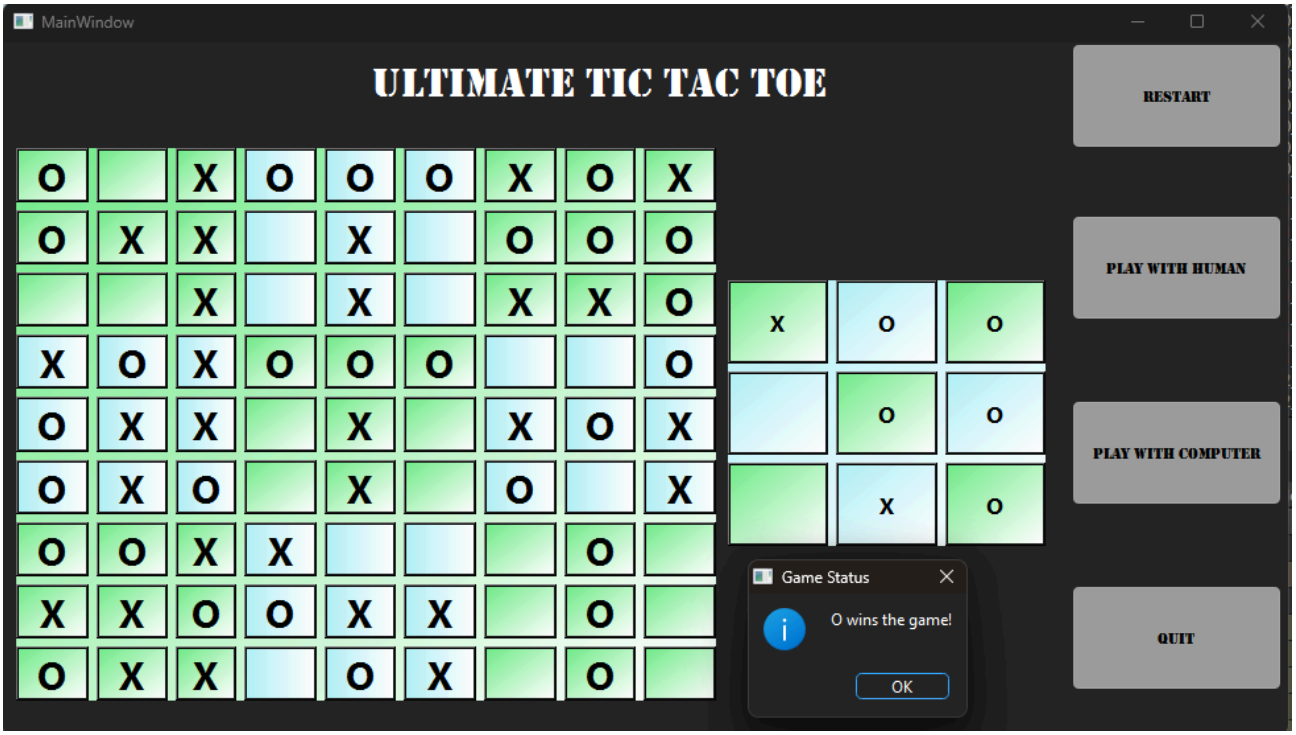
Github repo screenshot:



UML Diagram:


SUS_Random_Player
- IndexTranslator(x : int, y : int, index : short) : void
+ SUS_Random_Player(n : string, symbol : T)
+ getmove(x : int, y : int) : void
RandomPlayer
dimensions : int
+ RandomPlayer(symbol : T)
+ get_move(x : int, y : int) : void
Player
name : string
symbol : T
boardPtr : Board<T>
+ Player(n : string, symbol : T)
+ Player(symbol : T)
+ getmove(x : int, y : int) : void
+ get_symbol() : T
+ get_name() : string
+ setBoard(b : Board<T>) : void
Board
rows : int
columns : int
n_moves : int
board : T**
+ update_board(x : int, y : int, symbol : T) : bool
+ display_board() : void
+ is_win() : bool
+ is_draw() : bool
+ game_is_over() : bool
SUS_Board
- sCount : short
- uCount : short
- checks : vector<bool>
- display() : void
- WinningCount(count : short) : void
+ SUS_Board()
+ update_board(x : int, y : int, symbol : T) : bool
+ display_board() : void
+ is_win() : bool
+ is_draw() : bool
+ game_is_over() : bool
Ultimate_X_O_Random_Player
+ Ultimate_X_O_Random_Player(symbol : T)
+ getmove(x : int, y : int) : void
SUS_Player
- IndexTranslator(x : int, y : int, index : short) : void
- digitInput(temp : string) : bool
- can_numbers(n : short) : void
- checkValidIndex(index : short) : void
+ SUS_Player(n : string, symbol : T)
+ SUS_Player(symbol : T)
+ getmove(x : int, y : int) : void
Ultimate_X_O_Player
+ Ultimate_X_O_Player(name : string, symbol : T)
+ getmove(x : int, y : int) : void
Ultimate_X_O_Board
+ Final_board : vector<vector<T>>
+ current_symbol : char
+ Ultimate_X_O_Board()
+ update_board(x : int, y : int, symbol : T) : bool
+ display_board() : void
+ is_win() : bool
+ is_small_win(x : int, y : int) : bool
+ is_draw() : bool
+ game_is_over() : bool

GUI



1. Technologies Used

- Qt Framework: For creating the graphical user interface, managing signals and slots for user interaction, and designing a responsive layout.
- C++: For implementing game logic, managing state transitions, and ensuring efficient performance.

2. Features

- **Interactive UI:** A 9x9 grid representing the game board divided into smaller 3x3 subgrids, along with buttons for restart and mode selection.
- **Human vs. Human and Human vs. Computer Modes:** Supports competitive gameplay between two players or against an AI.
- **Dynamic State Management:** Handles updates to the main board and subgrids after every move.
- **Automated Computer Random Moves:** Implements logic for the computer player.
- **Game Status Display:** Notifies players of the game's outcome (win, draw) with visual cues and pop-ups.
- **Reset Functionality:** Provides seamless reset for a new game.

3. Code Strengths

- **Modularity:** Functions are well-organized, making the code maintainable and scalable.
    - **Example:** Separate methods for resetting the game state (`resetGameState`), handling cell clicks (`handleCellClick`), and updating the main board (`updateMainBoard`).

- **Reusability:** The code uses generalized methods like `getButton`, `updateButtonText`, and `checkLine` to avoid duplication.
- **UI Integration:** Efficient use of Qt's `findChild` and signal-slot mechanism to connect UI elements to the underlying logic.
- **Error Handling:** Safeguards against invalid moves, such as preventing plays in completed subgrids and validating user input.
- **Readability:** Well-commented code and clear variable names like `currentPlayer`, `smallBoardStatus`, and `mainBoard` enhance understanding.

4. Challenges and Bug Fixes

- **Small Board State Management:**
  - Issue: Mismanagement of subgrid states after a player wins or draws in that subgrid.
  - Solution: Introduced `smallBoardStatus` to track subgrid outcomes and prevent further moves in completed subgrids.
- **Edge Cases in Computer Logic:**
  - Issue: Computer did not block certain winning moves by the opponent.
  - Solution: Enhanced `computerMove` to simulate and evaluate potential opponent wins before making its move.
- **Subgrid-to-Main Grid Transition:**
  - Issue: Main board state was not updating correctly after a subgrid win.
  - Solution: Ensured proper propagation of subgrid outcomes to the main board through `updateMainBoard` and `updateFinalBoard`.
- **Draw Condition Handling:**
  - Issue: Draw conditions were not correctly identified for the main board.
  - Solution: Implemented `isSmallBoardDraw` and `is_main_draw` to accurately detect and handle draw scenarios.
- **UI Button State Management:**
  - Issue: Buttons were not being properly disabled after game completion or subgrid wins.
  - Solution: Added logic in `disableSmallBoard` and `resetGameState` to manage button states dynamically.

5. Code Quality Metrics

- **Performance:** The use of `QPushButton` references and efficient board traversal ensures smooth gameplay, even on larger grids.
- **Scalability:** The modular approach allows for easy extensions, such as additional game modes or larger board sizes.
- **Error-Free Execution:** Comprehensive testing ensures robust handling of edge cases, such as invalid moves, Computer conflicts, and simultaneous win conditions.
- **User Experience:** Clear feedback mechanisms like message boxes and button states enhance user interaction.

Quality of code:

George games:

Pyramid-Tic-Tac-Toe.h:

Strengths

1. Good Design:
    - Effective use of object-oriented principles.
    - Logical modularity with class-based structure.
2. AI Logic:
    - Implements MiniMax algorithm for decision-making.

Issues and Fixes

1. Memory Leaks:
    - Dynamic arrays lack proper cleanup.
    - Suggestion: Use `std::vector` or add destructors.
2. Repetition:
    - Hardcoded logic in `IndexTranslator`.
    - Suggestion: Replace with formula-based calculation.
3. Global Variable:
    - `AI` is globally defined.
    - Suggestion: Encapsulate in AI player class.
4. Insufficient Comments:
    - Critical logic lacks explanation (e.g., MiniMax).
    - Suggestion: Add clear, concise comments.

Word-Tic-Tac-Toe.h:

Strengths

1. Good Design:
    - Effective use of templates for flexibility with different data types.
    - Clear class structure separating responsibilities between `Word_Board`, `Word_Player`, and `Word_Random_Player`.
2. Input Handling
    - Proper validation of user input for characters and numbers.

: weaknesses

1. Hardcoded Ranges:
    - Hard Coded dictionary ranges in `wordChecker`.
    - Suggestion: Use a map or more efficient search algorithm for scalability and flexibility.
2. Unnecessary Pointers:
    - Pointers used for simple types like `short` and `char`. Suggestion: Replace with automatic variables.

SUS.h

Strengths

Good Design and input handling

Weaknesses

Global Variable:

- The `END` variable is global.
  Suggestion: Encapsulate within the `SUS_Board` class.

Kerolus Games

Four-in-a-row.h

Strengths

- Board Management: Handles dynamic board size and tracks moves.
- Game Logic: Effective win conditions (horizontal, vertical, diagonal), and draw check.

Issues and Fixes

- Memory Management: Use `std::vector` instead of `new` for better memory management.
- Display Logic: Improve formatting for clearer board display.
- Input Validation: Add checks in `getmove` for valid column range and non-numeric input
- Repeated Logic: Abstract move selection in `getmove` functions to avoid duplication.
- No Error Handling for Random Player: Ensure valid random column selection.

Numerical-Tic-Tac-Toe.h

Strengths

- Player Management: `NUM_X_O_Player` can keep track of available numbers for each player.
- Random Player: Random player generation works with valid symbols from available numbers.

Issues and Fixes

- Input Validation: Improve `getmove` in `NUM_X_O_Player` to handle invalid inputs and edge cases more effectively.
- Repeated Logic: Reduce code duplication, particularly in win condition check

- Error Handling: Consider more robust error handling when checking the player's available numbers.
- Board Display: Formatting of the board can be improved for better clarity.
- Unused `symbol` Field: The `symbol` field in `NUM_X_O_Player` is reassigned without being properly used for game logic (remove or clarify its role).

Marco Games

5x5-Tic-Tac-Toe.h

Strengths

- Error Handling: The input validation in `getmove` handles invalid entries and out-of-bound values.
- Random Player: The random player generates valid moves within the 5x5 grid.

Issues

- Global Variables: `counter_x` and `counter_o` are declared globally, which might lead to issues in larger programs.
- Hardcoded Values: The board size (5x5) and the move checks are hardcoded, making the code less flexible.
- Game Over Logic: The `is_win` method checks for a win based on the number of lines, but it could be more efficient in terms of readability and performance.
- Repetitive Logic: The line-checking logic for both diagonal directions and rows/columns is duplicated, which could be refactored.

Misere-Tic-Tac-Toe.h

Strengths

- Game Flow Control: The `move_tracker` variable efficiently controls the flow of the game by preventing further moves once the game is over.
- Input Validation: The `getmove` method ensures valid player input, and moves are rejected if the game is over.

Issues

- Global Variable: `move_tracker` is a global variable, which could lead to issues in a larger program or when handling multiple game instances. Consider encapsulating it within the relevant class.
- Game Over Logic: In the `is_win` method, the condition `move_tracker = this->n_moves + 3;` seems to be an unconventional way of marking a win. It could be improved by directly returning a win state.
- Board Memory Management: Using raw pointers for the board could potentially lead to memory leaks. Consider using `std::vector` to manage the board memory more safely.
- Hardcoded Values: The 3x3 board is hardcoded, which limits the flexibility of the game. Making the board size configurable could make it more reusable.

Kerolus and Marco game

Ultimate-Tic-Tac-Toe.h

Strengths:

1. Game Logic: The game properly simulates the rules of Ultimate Tic-Tac-Toe, including the nested board structure and win-checking mechanisms.
2. Clear Display: The `display_board` method is well-organized, showing both the main board and smaller 3x3 grids, making the game's state easy to understand.

Weaknesses:

1. Win Detection Complexity: The current win-checking logic is somewhat convoluted, making it hard to maintain and debug. Simplifying this logic could improve readability and performance.
2. Input Validation: There is no clear mechanism to ensure that user input is valid (e.g., checking for already-occupied spots or invalid grid positions).
3. Random Player Logic: The random player may attempt invalid moves (e.g., on an already occupied spot). This needs to be addressed by adding validation in the `getmove` function.
4. Scalability: The code is tightly coupled to a 9x9 grid with a specific 3x3 sub-grid structure. Extending it to larger or smaller boards would require significant changes to the codebase.

Classes Description

KMG Manager

The `Program` class acts as the main driver for a collection of board games implemented in the project. It provides a user interface for selecting and running different game modes and manages the overall flow of the application. The primary features of the class include:

1. **Header and Footer:**
    - **The `Header` and `exit` methods display introductory and exit messages, respectively, providing a polished user experience.**
2. **Main Menu:**
    - **The `MainMenu` method presents a list of available games for the user to choose from. Options include various Tic-Tac-Toe variations, "Four-in-a-row," and other specialized games like "SUS."**
3. **User Input Handling:**
    - **Methods like `digitInput`, `cin_numbers`, and `playerName` ensure robust input handling for numerical entries and player names.**
    - **The `playerType` method allows the user to select the type of players (e.g., human, random AI, unbeatable AI).**
4. **Game Initialization:**
    - **The `GameData` method gathers player names and types while allowing for AI selection if supported by the game.**
5. **Game Execution:**
    - **The `GameRunning` method is the core of the class. It:**
        - **Handles the user's menu selection.**
        - **Initializes the appropriate game class (e.g., `Pyramid_Board`, `Four_Board`).**
        - **Configures player objects (human or AI).**
        - **Utilizes a `GameManager` instance to start and manage gameplay.**
        - **Cleans up dynamically allocated resources to prevent memory leaks.**
6. **Supported Games:**
    - **Games are divided by contributors and their respective implementations:**
        - **Kerolos's files: "Four-in-a-row" and "Numerical Tic Tac Toe."**
        - **Marco's files: "5x5 Tic Tac Toe" and "Misere Tic Tac Toe."**
        - **George's files: "Pyramid Tic Tac Toe," "Word Tic Tac Toe," and "SUS."**
        - **Kerolos and Marco: "Ultimate Tic Tac Toe."**

7. **Memory Management:**
    - **Dynamic memory allocation for game boards and players is handled with `new` and cleaned up with `delete` to avoid resource leaks.**
8. **Exit Mechanism:**
    - **The user can quit the program by selecting option `0` from the menu.**

Pyramid-Tic-Tac-Toe Game

1. **`Pyramid_Board` Class:**
    - **Represents a 3x3 pyramid-style board for the game.**
    - **Handles the display and internal state of the board.**
    - **Includes methods to:**
        - **Display the board in a pyramid layout.**
        - **Update the board based on player moves.**
        - **Check for game-ending conditions like win or draw.**
    - **Initializes the board with numbers `1-9` for easy indexing and user interaction.**
2. **`Pyramid_Player` Class:**
    - **Represents a human player in the game.**
    - **Provides functionality to translate a single number input (1-9) into 2D board coordinates.**
    - **Ensures player input is valid (numeric and within range).**
    - **Includes methods to prompt and retrieve a valid move from the player.**
3. **`Pyramid_Random_Player` Class:**
    - **Represents an AI player that makes random valid moves.**
    - **Extends the functionality of `RandomPlayer<T>` with additional customization.**
    - **Uses a random number generator (`rand()`) to pick a move from the available positions.**
    - **Converts the randomly generated index into board coordinates using `IndexTranslator`.**
4. **`Pyramid_AI_Player` Class:**
    - **Represents an AI player that makes optimal moves using the MiniMax algorithm.**
    - **Implements a recursive `calc_MiniMax` function to evaluate all possible board states.**
    - **Uses the MiniMax strategy to maximize its chances of winning while minimizing the opponent's.**
    - **Ensures that the AI always selects the best possible move.**
    - **Includes methods to:**
        - **Simulate future moves and evaluate outcomes.**
        - **Determine the best move based on the board's current state.**

Four-in-a-row Game

1. `Four_Board` Class:
    - Represents the 6x7 grid used in the "Four in a Row" game.
    - Manages core board operations, including placing pieces, displaying the board, and checking for game conditions like a win or draw.
    - Key Methods:
        - `update_board`: Updates the board with a player's symbol at the lowest available row in a specified column.
        - `display_board`: Visually renders the game board, displaying column indices and player moves.
        - `is_win`: Checks rows, columns, and diagonals for four consecutive matching symbols, signaling a win.
        - `is_draw`: Determines if the board is full (42 moves) without a winner.
        - `game_is_over`: Combines `is_win` and `is_draw` to check if the game should end.
2. `Four_Player` Class:
    - Represents a human player.
    - Extends the `Player` base class, allowing users to input their moves.
    - Key Method:
        - `getmove`: Prompts the player to specify a column (0–6) for their move. Validates the column input.
3. `Four_Random_Player` Class:
    - Represents a computer-controlled player that makes random moves.
    - Extends the `RandomPlayer` base class and introduces randomness in selecting columns.
    - Key Method:
        - `getmove`: Randomly selects a column (0–6) for its move. Ensures variability in gameplay and provides a simple AI opponent.

5x5-Tic-Tac-Toe Game

1. `X_O_Board5` Class:
    - Represents a 5x5 board for the "5x5 Tic Tac Toe" game.
    - Extends the `Board` class and manages game logic, board updates, display, and victory conditions.
    - Key Features:
        - Initialization: Creates a 5x5 grid initialized with empty cells.
        - Move Management: Tracks moves and allows undo operations by resetting a cell.
        - Win Counting: Tracks and counts lines (rows, columns, diagonals) for each symbol (`X` or `O`).

        - Victory Conditions: Determines if a player wins based on the number of lines formed and the total moves.
    - Key Methods:
        - `update_board`: Updates a cell with a player's symbol or undoes a move by resetting the cell.
        - `display_board`: Displays the current board state, showing coordinates and cell values.
        - `count_lines`: Counts lines of three consecutive symbols for a given player.
        - `is_win`: Checks if a win condition is met based on move count and line counts.
        - `is_draw`: Checks if the game ends in a draw after 25 moves without a winner.
        - `game_is_over`: Combines `is_win` and `is_draw` to check if the game should end.
2. `X_O_Player5` Class:
    - Represents a human player in the "5x5 Tic Tac Toe" game.
    - Extends the `Player` class and manages user input for gameplay.
    - Key Features:
        - Prompts the user to enter row and column coordinates for their move.
        - Validates input to ensure it falls within the board's boundaries.
    - Key Method:
        - `getmove`: Asks the player for their move, ensuring valid input before proceeding.
3. `X_O_Random_Player5` Class:
    - Represents a computer-controlled player that makes random moves on the 5x5 board.
    - Extends the `RandomPlayer` class, introducing randomness to simulate simple AI behavior.
    - Key Features:
        - Randomly generates valid moves within the 5x5 grid.
        - Can act as an opponent for a human player or another random player.
    - Key Method:
        - `getmove`: Generates a random row and column coordinate for the move.

Word-Tic-Tac-Toe Game

1. `Word_Board` Class:
    - Inherits from a generic `Board` class.
    - Maintains a 3x3 game board and a dictionary of words loaded from a file (`dic.txt`).
    - Implements the core game logic:

    - **`uploadDictionary`: Reads the dictionary file into a vector of strings.**
    - **`wordChecker`: Checks if a combination of three characters forms a valid word in the dictionary by scanning predefined index ranges for each starting letter.**
    - **`display`: Renders the board with grid formatting.**
    - **Game state functions (`is_win`, `is_draw`, `game_is_over`) assess win/draw conditions based on word validation and the number of moves made.**
2. **`Word_Player` Class:**
    - **Represents a human player in the game.**
    - **Provides methods to translate user input into board indices and validate input for correctness.**
    - **Prompts the player to select a position and a character for their move.**
3. **`Word_Random_Player` Class:**
    - **A computer-controlled player that makes random moves.**
    - **Randomly selects a board position and character for its symbol using seeded random number generation.**
    - **Inherits from a generic `RandomPlayer` base class and appends "(Random Computer)" to the player's name for identification.**

Numerical-Tic-Tac-Toe Game

1. NUM_X_O_Board

- **Purpose: Implements the game board for NUM_X_O, a variant of Tic-Tac-Toe where the goal is to get a sum of 15 in any row, column, or diagonal using numbers instead of X's and O's.**
- **Constructor: Initializes a 3x3 board with all cells set to 0, representing empty spaces, and initializes the number of moves (`n_moves`) to 0.**
- **`update_board(int x, int y, T symbol)`: Validates and updates the board at coordinates (x, y) with the given symbol (number). The method ensures that the move is valid, i.e., the cell is empty or the number is being removed.**
- **`display_board()`: Displays the current state of the board, showing each cell's coordinates and its value.**
- **`is_win()`: Checks if any row, column, or diagonal has a sum of 15, indicating a win in the game.**
- **`is_draw()`: Returns `true` if all 9 cells are filled and there is no winner, meaning the game ends in a draw.**
- **`game_is_over()`: Returns `true` if either `is_win()` or `is_draw()` is `true`, indicating that the game has ended.**

2. NUM_X_O_Player

- **Purpose: Represents a human player in the NUM_X_O game, where they interact with the game by making moves using numbers.**
- **Constructor: Initializes the player with a name, symbol (the number they use), and a list of available numbers they can use during the game.**
- **`getmove(int& x, int& y)`: Prompts the player for a move, consisting of coordinates (x, y) and a number. It checks if the number is valid (i.e., still available for use) and updates the board if valid.**
- **`removeNumber(int number)`: Removes the specified number from the list of available numbers.**
- **`hasNumber(int number)`: Checks if the player still has the given number in their list of available numbers.**

3. NUM_X_O_Random_Player

- **Purpose: Represents a random computer-controlled player in the NUM_X_O game who makes moves without any strategy.**
- **Constructor: Initializes the random player with a symbol and a list of available numbers. It also sets a name for the player ("Random Computer Player") and seeds the random number generator for move selection.**
- **`getmove(int& x, int& y)`: Picks random coordinates for the move and randomly selects a number from the list of available numbers to place on the board.**

Misere-Tic-Tac-Toe Game

1. Misere_Board

- **Purpose: Implements the game board for the Misere variant of Tic-Tac-Toe. In Misere Tic-Tac-Toe, the objective is to avoid winning, with players trying to force their opponent into a winning position.**
- **Constructor: Initializes a 3x3 board with empty cells (value 0) and a move count (`n_moves`) set to 0.**
- **Methods:**
  - **`update_board(int row, int col, T symbol)`: Updates the board with the specified symbol (converted to uppercase) at the given coordinates, unless the game is already over.**
  - **`display_board()`: Displays the current state of the board, including coordinates and values for each cell.**
  - **`is_win()`: Checks if there is a winning combination on the board (any row, column, or diagonal with the same symbol).**
  - **`is_draw()`: Returns true if the game is a draw (9 moves made with no winner).**
  - **`game_is_over()`: Returns true if the game is over, either by a win or a draw.**

2. MiserePlayer

- **Purpose: Represents a human player in the Misere Tic-Tac-Toe game.**
- **Constructor: Initializes the player with a name and a symbol (the mark they will use on the board).**
- **Methods:**
  - `getmove(int& row, int& col)`**: Prompts the player to input their move (row and column). The player can input coordinates for their next move unless the game is already over.**

3. MisereRandom_Player

- **Purpose: Represents a computer-controlled player that makes random moves in the Misere Tic-Tac-Toe game.**
- **Constructor: Initializes the random player with a symbol and sets up the game dimension (3x3). The random number generator is seeded for making random moves.**
- **Methods:**
  - `getmove(int& row, int& col)`**: Generates a random move by selecting a random row and column within the board's dimensions.**

Ultimate-Tic-Tac-Toe Game

1. Ultimate_X_O_Board

- **Purpose: Implements the game board for Ultimate Tic-Tac-Toe, where each move dictates a smaller 3x3 grid in a larger 9x9 grid. The objective is to win in any of the 3x3 sub-grids or the overall 9x9 grid.**
- **Constructor: Initializes a 9x9 board with all cells set to 0, representing empty spaces. It also initializes a `final_board` to track the winners of the 3x3 sub-grids and sets the number of moves (`n_moves`) to 0.**
- `update_board(int x, int y, T mark)`**: Validates and updates the board at coordinates (x, y) with the given symbol (mark). It ensures that the move is valid, i.e., the cell is empty, the correct sub-grid is available, and updates the move count accordingly.**
- `display_board()`**: Displays the current state of both the 9x9 board and the `final_board`, showing each cell's coordinates and its value. The main board is divided into smaller 3x3 sections for clarity.**
- `is_small_win(int x, int y)`**: Checks if there's a winner in the smaller 3x3 grid starting at coordinates (x, y). A win is determined by checking rows, columns, and diagonals.**
- `is_win()`**: Checks if any of the 3x3 sub-grids have a winner and updates the `final_board` accordingly. Then, it checks for a winner in the 3x3 `final_board` by checking rows, columns, and diagonals.**
- `is_draw()`**: Returns `true` if all cells in the 9x9 grid are filled and no winner has been determined, indicating a draw.**

- **`game_is_over()`: Returns `true` if either `is_win()` or `is_draw()` returns `true`, indicating that the game has ended.**

2. Ultimate_X_O_Player

- **Purpose: Represents a human player in the Ultimate X-O game. The player interacts with the game by making moves on the 9x9 grid using a specific symbol (either 'X' or 'O').**
- **Constructor: Initializes the player with a name and symbol (mark).**
- **`getmove(int& x, int& y)`: Prompts the player for a move, consisting of coordinates (x, y), and checks whether the move is valid before updating the board.**

3. Ultimate_X_O_Random_Player

- **Purpose: Represents a random computer-controlled player in the Ultimate X-O game, making moves without any specific strategy.**
- **Constructor: Initializes the random player with a symbol (mark) and a name ("Random Computer Player"). Seeds the random number generator for move selection.**
- **`getmove(int& x, int& y)`: Selects random coordinates on the 9x9 board and randomly chooses a symbol to place on the board.**

SUS Game

1. SUS_Board Class

- **Purpose: Implements a 3x3 game board for a variant of Tic-Tac-Toe with custom rules. It manages moves, checks for winning conditions, and displays the board.**
- **Constructor: Initializes the board with numbers 1 to 9 and sets up necessary attributes like move counts and a boolean vector (`checks`) for tracking winning conditions.**
- **`update_board(int x, int y, T symbol)`: Updates the board at the given coordinates with the specified symbol (`S` or `U`), ensuring valid moves (numbers 1-9).**
- **`display_board()`: Displays the current state of the board with all cell values.**
- **`is_win()`: Checks for a win based on specific conditions defined in `WinningCount()`.**
- **`is_draw()`: Returns `true` if the game ends in a draw.**
- **`game_is_over()`: Returns `true` if the game is over, either by a win or a draw.**

2. SUS_Player Class

- **Purpose: Represents a human player who makes moves by inputting numbers from 1 to 9.**
- **Constructor: Initializes the player with a name and symbol (`S` or `U`).**

- **`getmove(int &x, int &y)`: Prompts the player for a move, translating the input (1-9) into coordinates on the board.**
- **`digitInput(string &temp)`: Ensures the player enters a valid number between 1 and 9.**
- **`checkValidIndex(short &index)`: Ensures the move is within the valid range (1-9).**

3. SUS_Random_Player Class

- **Purpose: Represents a random computer player who makes moves without any strategy, selecting randomly from available positions.**
- **Constructor: Initializes the player with a name and symbol, and sets up random move generation.**
- **`getmove(int &x, int &y)`: Chooses a random move from available positions (1-9) and translates it to coordinates on the board.**
