

Cairo University

Faculty of Computers and Artificial Intelligence

**Object Oriented Programing**

**CS213**

**Assignment 2: Board Games**

Submitted to: **Dr. Mohammad El-Ramly**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Email** | **Section** | **ID** |
| Ahmed Mostafa Abdel-Kareem Sayed | 01029943015a@gmail.com | S12 | 20240822 |
| Belal Gamal Faris Abd Al Aziz | belalgamal131@gmail.com | S12 | 20230095 |
| Mohamed Ehab Sabry Saber | messk8@gmail.com | S12 | 20230547 |
| Mohammad Yahia Tammam | mohammad.yahia.tammam.20042010@gmail.com | S15 | 20235076 |

***Classes Description***

* **Pyramic Tic-Tac-Toe Classes**
  + **correct\_indecies Class:** This class manages the collection of valid move indices on the game board.
    - **Static Vector of Pairs**: It contains a static vector<pair<int, int>>\* correct\_index that holds valid coordinates for moves on the board.
    - **Constructor**: The constructor initializes the correct\_index if it is not already set, using a predefined set of valid coordinates (pairs of integers).
    - **Destructor**: The destructor deletes the correct\_index pointer if it has been initialized, cleaning up memory to prevent leaks.
  + **Triangle\_XO\_Board Class:** This class represents the game board for the Triangle XO game. It inherits from Board<char> and correct\_indecies.
    - **Constructor**: It sets the board size to 3 rows and 5 columns and initializes the board with # (empty space).
    - **update\_board Method**: This method takes an x and y coordinate, checks if the move is valid (by looking up the index in correct\_indecies), updates the board if valid, and returns true. If the index is invalid, it returns false.
    - **display\_board Method**: This method prints the current state of the game board to the console.
    - **is\_win Method**: It checks if there is a winning condition on the board (either horizontally, vertically, or diagonally).
    - **is\_draw Method**: It checks if the game is a draw (i.e., no winner and all moves are exhausted).
    - **game\_is\_over Method**: It returns true if the game is either won or drawn, signaling the end of the game.
  + **Triangle\_XO\_player Class**: This class represents a player in the Triangle XO game. It inherits from Player<char> and correct\_indecies.
    - **Constructor**: It has two constructors. One for a human player with a name and symbol, and another for a computer player with only a symbol.
    - **getmove Method**: It prompts the player to enter a valid move (coordinates). If an invalid index is entered, it asks the player to enter the correct index.
    - **getName Method**: Returns the player's name.
    - **getSymbol Method**: Returns the player's symbol (X or O).
  + **Triangle\_XO\_Random\_player Class:** This class represents a random AI player for the Triangle XO game. It inherits from RandomPlayer<char> and correct\_indecies.
    - **Constructor**: It initializes the random player with a symbol (X or O) and sets the player's name to "Artificial stupidity".
    - **getmove Method**: It selects a random valid move from the available valid indices and updates the x and y values with that move.
  + **Triangle\_XO Class:** This is the main game controller class that orchestrates the entire game.
    - **readString Method**: It reads a string input from the user, including spaces.
    - **ReadNumber Method**: Ensures that the user input is a valid number.
    - **readNumInRange Method**: Ensures that the input is a valid number within a specified range.
    - **take\_players\_info Method**: Prompts the user for player names and types (Human or Random AI) for both players (X and O).
    - **start\_Triangle\_XO Method**: This method starts the game by setting up the two players based on user input, initializing the game board, and managing the game flow using the GameManager.
* **Four-in-a-row Classes**
  + **Connect4 Class:** Inherits from Board<char>, which is presumably a template class handling board operations. The board is represented as a 6x7 grid, and each cell can hold a character ('X', 'O', or empty).
    - **Private Members:**
    - **vector<vector<char>> board:** The actual game board, a 2D grid where each position can store a player's piece.
    - **map<int, int> howFull:** Tracks how full each column is (the number of pieces already placed in each column).
    - **Public Members:**
    - **Constructor (Connect4):** Initializes the board with 6 rows and 7 columns, and sets all positions to empty (' ').
    - **display\_board:** Displays the current state of the board in a user-friendly format with bold and colored text.
    - **getN\_moves:** Returns the total number of moves made so far in the game.
    - **check\_status:** Determines the current state of the game (e.g., win, draw, ongoing) and returns an integer to represent the state.
    - **is\_win:** Checks if there is a winning condition (4 consecutive symbols horizontally, vertically, or diagonally).
    - **is\_draw:** Checks if the game is a draw (42 moves without a winner).
    - **game\_is\_over:** Returns true if the game is over (either won or drawn).
    - **update\_board:** Updates the board with a player's move if the column is not full and the move is valid.
    - **getHowFull:** Returns the howFull map to access the current status of the columns.
  + **C4\_Player Class:** Inherits from Player<char>, which represents a player in the game. This class would manage player-specific attributes and actions.
    - **Private Members:**
    - **Connect4 \*c4\_p:** A pointer to a Connect4 object, which represents the actual game board for the player.
    - **Public Members:**
    - **Constructor (C4\_Player):** Initializes the player with a name and symbol (either 'X' or 'O') and associates the player with a Connect4 board.
    - **getmove:** Prompts the player to choose a column for their move. It ensures the input is valid (between 1 and 7).
    - **getsymbol:** Returns the player's symbol ('X' or 'O').
  + **C4\_Random\_Player Class:** Base Class: Inherits from Player<char>, like C4\_Player, but represents a random AI player.
    - **Private Members:**
    - **int dimension:** The number of columns in the Connect Four board (7).
    - **Public Members:**
    - **Constructor (C4\_Random\_Player):** Initializes the player with a name and symbol, and seeds the random number generator.
    - **getmove:** Randomly selects a column (between 1 and 7) for the AI's move.
  + **playConnect4 Function:** This function is the entry point for the game. It:
    - prompts the user to enter player names and choose the type of players (human, random AI, or potentially smart AI in the future).
    - Creates the Connect4 game board and initializes the players (either human or AI).
    - Uses a GameManager<char> (presumably another class not shown in this code) to manage and run the game.
    - After the game is over, it cleans up the dynamically allocated memory for the game board and players.
* **5 x 5 Tic-Tac-Toe Classes**
  + **TicTacToe5x5 Class:** This class represents the main logic for a 5x5 Tic-Tac-Toe game. It inherits from a base class Board<T>, suggesting that it's built upon a general board game class.
    - **Member Variables:**
    - **vector<vector<char>> v**: This 2D vector represents the game board, with each cell being a character. It's initialized to a 7x7 board (though only the 5x5 inner part is used) to handle boundary conditions more easily.
    - The extra rows and columns are likely to help with easier indexing and boundary handling.
    - The game board uses ~ as the placeholder for empty spaces.
    - **char start = '~'**: This character is used to represent empty spots on the board.
    - **int n\_moves = 0**: Keeps track of how many moves have been made during the game. It's useful for determining when the game is over (either due to a win or a draw).
    - **Methods:**
    - **TicTacToe5x5()**: Constructor that initializes the 7x7 board. The outer cells are extra padding to make the logic easier, while the game uses a 5x5 subset of the grid for actual play.
    - **bool update\_board(int x, int y, T symbol)**: Updates the board at the specified row (x) and column (y) with the symbol (either X or O).
    - It checks whether the coordinates are valid (between 1 and 5) and if the spot is empty. If either condition fails, it outputs an error message and returns false.
    - **void display\_board()**: Displays the board with color formatting. The grid is printed row-by-row, and each symbol is colored differently (X in red, O in green, and empty spaces in cyan).
    - The score of both players (X and O) is also displayed.
    - **bool is\_win()**: Checks if the game has a winner. The game is considered won if a player has more points than the other, where points are calculated by the score() method.
    - If X has more points, Player X wins; if O has more, Player O wins. If both players have equal scores, it results in a draw.
    - **bool is\_draw()**: Checks if the game is a draw. This occurs when the total number of moves reaches 24 (i.e., all positions are filled) and there is no winner.
    - **bool game\_is\_over()**: Returns true if the game is over, either due to a win or a draw.
    - **int score(const vector<vector<char>>& v, const char x)**: Calculates the score for a given player (either X or O).
    - It iterates through the board and checks for lines of adjacent matching symbols (horizontally, vertically, and diagonally). If such lines are found, points are awarded for those lines.
    - **Purpose**
    - The TicTacToe5x5 class handles the core functionality of the game, including board management, player moves, win conditions, and drawing the game state.
  + **tic\_tac\_Player Class:** This class represents a human player in the Tic-Tac-Toe game. It inherits from a generic Player<T> class (not shown in the provided code, but presumably a base class for different types of players).
    - **Constructor**
    - **tic\_tac\_Player(string name, T symbol)**: The constructor takes the player’s name and symbol (either X or O) and passes them to the base Player<T> class. This sets up the player's identity and symbol.
    - **Methods**
    - **void getmove(int& x, int& y)**: This method prompts the user for their next move (row and column) and validates the input.
    - If the input is invalid (non-numeric, out of bounds, etc.), the method keeps asking until valid input is received.
    - **Purpose**
    - The tic\_tac\_Player class is used to represent a human player who interacts with the game by providing moves through the console.
  + **tic\_tac\_Random\_Player Class:** This class represents a random computer player. It also inherits from RandomPlayer<T>, a class likely providing base functionality for random players.
    - **Constructor**
    - **tic\_tac\_Random\_Player(T symbol)**: This constructor initializes the random player with a given symbol (X or O) and sets a default name ("Random Computer Player").
    - It also seeds the random number generator using srand and the current time (time(0)), ensuring that the random moves differ between game sessions.
    - **Methods**
    - **void getmove(int& x, int& y)**: This method generates random valid moves within the board’s 5x5 dimension.
    - It generates random x and y values within the range of 1 to 5 (inclusive) using the rand() function.
    - **Purpose**
    - The tic\_tac\_Random\_Player class provides functionality for a random AI opponent that picks moves without any strategy. It is an easy way to introduce computer opponents into the game, but it does not have any advanced strategy for winning.
  + **playTicTacToe5x5 Function:** This is a utility function to start and run the 5x5 Tic-Tac-Toe game.
    - **Logic:** The function first asks for Player X’s name and whether they want to play as a human, random AI, or another type of AI (though AI is not implemented in the provided code). Similarly, it asks for Player O’s name and type.Based on the choices, it creates instances of either tic\_tac\_Player, or tic\_tac\_Random\_Player for both players. A GameManager<char> (likely a class for managing the game loop) is created and run.The function then deletes the dynamically allocated memory for the game board and players after the game is over.
    - **Purpose**
    - This function is the main entry point for running the game. It handles user input, game setup, and clean-up, effectively running the game loop.
* **Word Tic-tac-toe**
  + **correct\_indexies Class:**
    - **Purpose:** Stores the valid indices for the Tic-Tac-Toe board.
    - **Static Member:** correct\_indexs is a static pointer to a vector<pair<int, int>> which holds the valid pairs of coordinates on the board.
    - **Constructor:** If correct\_indexs is not initialized, it initializes it with a list of valid indices representing the positions on a 3x3 board.
    - **Destructor:** Deletes the correct\_indexs vector to prevent memory leaks.
  + **words Class:**
    - **Purpose:** Manages a set of words loaded from a dictionary file (dic.txt).
    - **Static Member:** all\_words is a static pointer to a set<string>, which holds the words read from dic.txt.
    - **Constructor:** If all\_words is not initialized, it opens the dic.txt file, reads each line, and adds it to the set<string>.
    - **Destructor:** Deletes the all\_words set to prevent memory leaks.
  + **Word\_XO\_Board Class:**
    - **Purpose:** Manages the Tic-Tac-Toe board and game logic.
    - **Inheritance:** It inherits from Board<char>, correct\_indexies, and words, making it aware of the board structure, valid indices, and the dictionary of words.
    - **Member Variables:**
    - **rows and columns:** define the size of the board (3x3).
    - **board:** is a dynamic 2D array of characters initialized with # to represent empty spaces.
    - **n\_moves:** tracks the number of moves made.
    - **Constructor:** Initializes the board and sets up the game state.
    - **Methods:**
    - **update\_board (int x, int y, char symbol):** updates the board at the specified position if the coordinates are valid, marking the move with the player's symbol.
    - **display\_board():** prints the current state of the board to the console.
    - **is\_win():** checks if a player has won by checking all rows, columns, and diagonals for a valid word (either forward or reversed) using the dictionary.
    - **is\_draw():** checks if the game is a draw (no winner and all spaces filled).
    - **game\_is\_over():** checks if the game has ended either by a win or a draw.
  + **Word\_XO\_player Class:**
    - **Purpose:** Represents a human player in the game, with the ability to make moves and select a symbol.
    - **Inheritance:** Inherits from Player<char> and correct\_indexies.
    - **Member Variables:**
    - **name:** stores the player's name.
    - **Methods:**
    - **ReadNumber():** is a helper function to ensure the input is a valid number.
    - **getmove(int& x, int& y):** allows the player to input the coordinates for their move. It checks if the input coordinates are valid and asks for re-entry if they're not.
    - **getName() and getSymbol():** return the player's name and symbol, respectively.
  + **Word\_XO\_Random\_player Class:**
    - **Purpose:** Represents a random computer player in the game.
    - **Inheritance:** Inherits from RandomPlayer<char> and correct\_indexies.
    - **Constructor:** Initializes the random player with a symbol and a preset name ("Artificial stupidity").
    - **Methods:**
    - **getmove(int& x, int& y):** generates random valid coordinates for the move and selects a random symbol for the move.
  + **Word\_XO Class:**
    - **Purpose:** Manages the flow of the Tic-Tac-Toe game, including player setup and managing game states.
    - **Member Variables:**
    - Contains several helper functions like readString(), ReadNumber(), and readNumInRange() to manage player input.
    - take\_players\_info() handles the setup of both players, allowing the user to choose between human and computer players for both 'X' and 'O'.
    - **Methods:**
    - **start\_Triangle\_XO():** starts the game by initializing the players based on user input and invoking the GameManager to run the game.
    - **Key Functionalities:**
    - **Game Flow:**
    - Players (either human or random AI) take turns selecting valid coordinates to place their marks ('X' or 'O').
    - The game checks after each move whether there is a winner or if the game is a draw.
    - **Dictionary Check for Win:**
    - A unique feature in this implementation is the use of the dictionary (dic.txt) to check if a row, column, or diagonal forms a valid word (either forward or reversed). This extends the typical Tic-Tac-Toe by allowing the winning conditions to be based on words.
    - **Dynamic Memory Management:**
    - Several classes use dynamic memory (new and delete) to manage resources like the game board, valid indices, and the dictionary of words.
* **Numerical Tic-Tac-Toe Classes**
  + **Numerical\_Board Class:** This class represents the game board and encapsulates the logic for managing and displaying the board for a numerical Tic-Tac-Toe game, using a 3x3 grid. It also includes the game state management and logic for making moves.
    - **Key Attributes:**
    - **board:** A 2D vector representing the 3x3 game board. Each element of the board holds an integer representing a player's choice (or 0 if the cell is empty).
    - **pl1\_available\_choices and pl2\_available\_choices:** These vectors store the available choices for player 1 and player 2, respectively. Player 1 can choose odd numbers (1, 3, 5, 7, 9) and player 2 can choose even numbers (2, 4, 6, 8).
    - **rows and columns:** Represent the size of the board (3x3 in this case).
    - **n\_moves:** Tracks the number of moves remaining in the game.
    - **Key Methods:**
    - **Numerical\_Board():** Constructor that initializes the board with 3x3 dimensions, sets the available moves for both players, and initializes the number of moves.
    - **~Numerical\_Board():** Destructor.
    - **display\_board():** Displays the current state of the board to the console, formatting the output with colors and bold/underline text for emphasis.
    - **getN\_moves():** Returns the number of remaining moves.
    - **MiniMax():** Implements the MiniMax algorithm for decision-making in AI players (discussed later). This method recursively evaluates all possible moves for the current player and returns the best one based on the game's rules.
    - **check\_status():** Checks if there’s a winner, a draw, or the game is still ongoing.
    - **update\_board():** Updates the game board after a player makes a move, ensuring the move is valid (coordinates are within bounds, and the chosen number is available for the player).
    - **is\_win():** Checks if a player has won the game by having a line (row, column, or diagonal) with the sum equal to 15.
    - **is\_draw():** Checks if the game is a draw, which happens if all moves are used without a winner.
    - **game\_is\_over():** Checks if the game is over, either by win or draw.
    - **Nume\_Player Class:** This class represents a human player in the numerical Tic-Tac-Toe game.
    - **Key Attributes:**
    - **Num\_board\_ptr:** A pointer to the Numerical\_Board instance that the player interacts with.
    - **Key Methods:**
    - **Nume\_Player():** Constructor that initializes the player with a name and symbol, and assigns a reference to the game board.
    - **operator<<:** Overloaded << operator for displaying the available choices for the player in a colored format.
    - **getmove():** Prompts the player for their move (coordinates) and validates the move using the isValidMove() function.
    - **getsymbol():** Asks the player to choose a symbol from the list of available choices and assigns it to the player. The available choices depend on whether it's player 1's or player 2's turn.
  + **Nume\_Random\_Player Class:** This class represents a random computer player in the numerical Tic-Tac-Toe game.
    - **Key Attributes:**
    - **dimension:** The dimension of the board (3x3 grid in this case, but the code was designed for 4x4 as well).
    - **Key Methods:**
    - **Nume\_Random\_Player():** Constructor that initializes the player with a random symbol.
    - **getmove():** Generates a random move by selecting random coordinates within the board’s bounds.
    - **getsymbol():** Selects a random symbol for the player, ensuring it is one of the available choices.
  + **MiniMax\_Player Class:** This class represents a player using the MiniMax algorithm for decision-making, essentially an AI player that makes optimal moves.
    - **Key Attributes:**
    - **value:** Stores the value of the move based on the evaluation from the MiniMax algorithm.
    - **numerical\_board:** A pointer to the Numerical\_Board instance that the AI interacts with.
    - **Key Methods:**
    - **MiniMax\_Player():** Constructor that initializes the player, sets their symbol to 0 (or another valid value), and creates a new Numerical\_Board instance.
    - **getmove():** Uses the getBestMove() method to calculate the best move for the AI based on the MiniMax algorithm.
    - **getBestMove():** Implements the MiniMax algorithm to find the best possible move for the AI by evaluating all possible moves on the board. The algorithm checks if a move will lead to a win or block the opponent from winning.
  + **GameManager Class:** (Not Fully Detailed in Code Above, But Inferred). This class likely handles the overall flow of the game, taking care of alternating turns between players, checking the game status, and determining the winner.
    - **Key Responsibilities:**
    - **Managing Turns:** Alternates between player 1 and player 2.
    - **Game Loop:** Runs the game, prompting players for their moves, updating the board, and checking if the game has ended (win or draw).
    - **Cleanup:** Deallocates any dynamically allocated memory after the game is finished.
  + **Board Class:** (Referenced but Not Fully Provided). The Board class is referenced as a parent class for the Numerical\_Board class. It likely contains basic board functionality that is inherited and extended by Numerical\_Board. This could include attributes for the board size, row and column counts, and possibly methods for common operations like initializing the board or displaying it.
    - **Summary of Gameplay Flow:**
    - **Player Setup:** Players can choose between being a human, a random computer, or an AI player (using the MiniMax algorithm).
    - **Turn-Based Gameplay:** Players take turns selecting a number and placing it on the board in a 3x3 grid, aiming to form a row, column, or diagonal with the sum equal to 15.
    - **Game End:** The game ends either when there is a winner (a player forms a line with a sum of 15) or a draw (no moves are left and no winner).
* **Misere Tic\_Tac\_Toe Class:**
  + **MisereTicTacToeBoard Class:**
  + **Purpose:** Represents the Misere Tic Tac Toe game board, with rules distinct from traditional Tic Tac Toe (losing if you create a winning line).
    - **Inherits:** From a generic Board<T> class.
    - **Key Members:**
    - v: A 5x5 vector grid initialized with ~.
    - start: Default character indicating an empty cell.
    - n\_moves: Counter for the number of moves made.
    - score: A helper function to calculate win conditions.
    - **Key Functions:**
    - update\_board(int x, int y, T symbol): Updates the board with the given symbol if the cell is valid and empty.
    - display\_board(): Prints the current game state, using colors to represent X, O, and empty cells.
    - is\_win(): Checks for a "loss" condition (the opponent forms a winning line).
    - is\_draw(): Verifies if the game is drawn (9 moves without a win).
    - game\_is\_over(): Combines is\_win() and is\_draw() to determine if the game has ended.
  + **MisereTicTacToePlayer Class:**
    - **Purpose:** Represents a human player.
    - **Inherits:** From Player<T>.
    - **Key Functions:**
    - Constructor initializes the player with a name and symbol (X or O).
    - getmove(int& x, int& y): Prompts the player for a move and validates it. Handles invalid inputs with error messages.
  + **MisereTicTacToeRandomPlayer Class:**
    - **Purpose:** Represents a random computer player.
    - **Inherits:** From RandomPlayer<T>.
    - **Key Functions:**
    - Constructor initializes the player with a random seed for move generation.
    - getmove(int& x, int& y): Generates a random move within the board dimensions.
    - **Helper Functions**
    - **score (in MisereTicTacToeBoard):**
    - Computes how many lines can still result in a win for a given symbol (X or O).
    - Uses 2D traversal to check horizontal, vertical, and diagonal continuity.
  + **playMisereTicTacToe()**
    - **Purpose:** Sets up and manages the game, including player selection.
    - **Steps:**
    - Prompts for player names.
    - Lets users choose player types (human, random AI, or smart AI).
    - Smart AI (tic\_tac\_MinMax\_Player) is commented out.
    - Initializes a GameManager object with the selected players and game board.
    - Runs the game using MiserTicTacToe.run().
* **Ultimate Tic\_Tac\_Toe**
  + **ultimate\_board<T> (template class):** This class represents the "Ultimate Tic-Tac-Toe" board and inherits from Board<T>. It has several methods for managing the game state and handling the board layout:
    - **X\_O\_Board<T> boards[10]:** An array of 10 X\_O\_Board objects, where each board represents a 3x3 grid, and the 10th board represents the overall game board.
    - **Constructor (ultimate\_board):** Initializes a 9x9 grid (as an array of char) for the game board, setting all cells to empty (' ') and initializing the move count to zero.
    - **is\_win():** Checks if the overall game is won (based on the 10th board).
    - **check\_Winner(int x, int y):** Determines the winner in a specific 3x3 sub-grid by checking rows, columns, and diagonals.
    - **is\_draw():** Checks if all the sub-boards are either won or drawn, and the game ends in a draw.
    - **display\_board():** Displays the current state of the entire 9x9 grid, showing both the sub-boards and the overall winner in the larger grid.
    - **update\_board(int x, int y, T mark):** Updates the main game board and the corresponding sub-board with the given mark (X or O).
    - **game\_is\_over():** Checks if the game is over due to a win or draw.
  + **ultimate\_player<T> (template class):** This class represents a player in the game and inherits from Player<T>.
    - **Constructor (ultimate\_player):** Initializes a player with a name and a symbol (either 'X' or 'O').
    - **getmove(int& x, int& y):** Prompts the player for input (row and column), validating the input to ensure it is numeric and within valid bounds (1-9). It keeps prompting until valid input is provided.
  + **ultimate\_Rplayer<T> (template class):** This class represents a "Random Player" (a computer player) and inherits from X\_O\_Random\_Player<T>.
    - **Constructor (ultimate\_Rplayer):** Initializes the random player with a name and symbol. It also seeds the random number generator with the current time.
    - **getmove(int& x, int& y):** Generates random row and column values within the grid bounds (1-9) to simulate a random move for the computer player.
  + **playUltimate() function:** This function handles the game setup and flow.It prompts the user to enter names for both players (X and O) and lets them choose whether each player will be human, a random computer player, or an AI-based player. Based on the players' choices, it creates instances of ultimate\_player<T> or ultimate\_Rplayer<T>.It initializes the game with a GameManager<char> object that handles the game loop and orchestrates player moves.Finally, it cleans up by deleting the dynamically allocated ultimate\_board and player objects.
* **SUS**
  + **SUS\_XO\_Board Class:** This class represents the board for the SUS XO game. It inherits from two classes:
    - **Board<char>**: A generic board class that uses characters as the game pieces.
    - **correct\_indexies**: This class seems to store valid (correct) indices for the game moves.
    - **Key Features:**
    - **Private Members:**
    - Scounter, Ucounter: These counters track the number of "SUS" (S) and "U" occurrences on the board.
    - **Constructor:**
    - Initializes a 3x3 board (board), where each position starts as # (unoccupied).
    - The number of moves (n\_moves) is initialized to 0.
    - **Methods:**
    - **num\_of\_SUS():** Searches for the occurrence of the word "SUS" horizontally, vertically, and diagonally on the board. It counts how many times the sequence "SUS" appears.
    - **update\_board(int x, int y, char symbol):** Updates the board with the given symbol (S or U) at the specified indices. It also checks if the move is valid and updates counters for "S" and "U" based on the number of "SUS" occurrences.
    - **display\_board():** Displays the current state of the board along with the counters for "S" and "U".
    - **is\_win():** Returns true if there is a winner (either "S" or "U").
    - **is\_draw():** Returns true if the game ends in a draw.
    - **game\_is\_over():** Returns true if the game is over (either a win or a draw).
  + **SUS\_XO\_player Class:** This class represents a player in the game. It also inherits from:
    - **Player<char>**: A generic player class that uses characters as the player's symbol.
    - **correct\_indexies**: Again, it uses this class for storing valid moves.
    - **Key Features:**
    - **Private Members:**
    - name: Stores the player's name.
    - **Constructor:**
    - Two constructors are available: one for a human player (with a specified name and symbol) and another for a computer player (with only the symbol).
    - **Methods:**
    - **ReadNumber(string message):** Reads a number from input and ensures it's a valid number.
    - **getmove(int& x, int& y):** Prompts the player to enter their move (row and column). It ensures the move is valid by checking if the indices are correct.
    - **getName() and getSymbol():** Return the player's name and symbol, respectively.
  + **SUS\_XO\_Random\_player Class:** This class represents a random computer player, which inherits from:
    - **RandomPlayer<char>**: A generic random player class that uses characters as the player's symbol.
    - **correct\_indexies**: Again, this is used for managing valid moves.
    - **Key Features:**
    - **Constructor:**
    - The constructor sets up a random number generator (srand), and initializes the player's name to "Artificial stupidity."
    - **Methods:**
    - **getmove(int& x, int& y):** Selects a random valid move from the available moves and updates the coordinates.
  + **SUS\_XO Class:** This is the main class for managing the SUS XO game. It handles the setup of the game and players, as well as running the game.
    - **Private Methods:**
    - **readString():** Reads a string input (with spaces allowed).
    - **ReadNumber():** Ensures that the input is a valid number.
    - **readNumInRange():** Ensures that the input number is within a specified range.
    - **take\_players\_info():** Gathers the necessary information from players, such as names and types (human or computer).
    - **Public Method:**
    - **start\_SUS\_XO():** This method starts the game by setting up the players based on user input, and then it initializes the game board and runs the game through the GameManager.

***The UML Class Diagrams:***

* **SUS Game:**

A screenshot of a computer

Description automatically generated

* **Ultimate Tic\_Tac\_Toe Game:**

A diagram of a game

Description automatically generated

***Work Distribution Table:***

|  |  |
| --- | --- |
| **Name** | **Work Distribution** |
| Ahmed Mostafa Abdel-Kareem Sayed | Games (3,6), Ultimate Tic\_Tac\_Toe Game, UML Design for SUS and Ultimate Tic\_Tac\_Toe games |
| Belal Gamal Faris Abd Al Aziz | Games (1,4), SUS Game |
| Mohamed Ehab Sabry Saber | Games (2,5), Menu and SUS Game GUI, Report formatting and adjustment. |
| Mohammad Yahia Tammam | Writing Classes descriptions |

***Code Reviews:***

**Code review by Ahmed Mostafa:**

**Game 1:**

The Triangle\_XO game logic is well-defined and implemented, but it lacks stakeholder approval. The code is well-formatted, but there are issues with excessive use of comments, inconsistent spacing in variable declarations, and unnecessary whitespace. Best practices include the Single Responsibility Principle (SRP), error handling, and minimal nesting.  
  
Maintainability can be improved by breaking long methods into smaller functions, simplifying expressions, and avoiding repetition. Memory management should be improved by using std::unique\_ptr or std::shared\_ptr for board and players, and replacing dynamic arrays with std::vector.  
  
Architecture issues include platform-dependent system clearing, no validation for user-provided names or symbols, and separation of concerns into classes. Configurable parameters like board size and player symbols are not configurable, and feature switching is lacking.  
  
Testing is missing, with no evidence of unit testing for methods like is\_win or update\_board. Edge cases are untested, and the code does not handle invalid memory allocation or invalid symbol inputs. Invalid inputs are validated through the getmove method, and inputs are sanitized through input validation.  
  
Documentation is sufficient, but some comments can be removed. There is no accompanying documentation or instructions for setting up and running the game. Overall, the Triangle\_XO game logic is functional as intended but requires improvement in readability, error handling, and documentation.

**Game 2:**

The code meets the basic requirements for implementing a Connect4 game, including board management, player actions, win/draw checking, and game loop handling. However, some features, such as the "Smart Computer (AI)" player, are commented out and not implemented. Stakeholder approval is not explicitly documented, so it is crucial to ensure approval if part of a larger project.  
  
The code is well-formatted, with consistent indentation and spacing. However, some redundant comments and unnecessary whitespace could be removed or clarified. Best practices include following the Single Responsibility Principle, handling different errors robustly, logging errors and warnings, avoiding magic values, omitting unnecessary comments, and using minimal nesting.  
  
Maintainability can be improved by extracting repeated logic, refactoring the is\_win() function into reusable functions, and breaking down lengthy methods into smaller helper functions for readability. Performance is acceptable for small board sizes, but diagonal checks in is\_win() iterate redundantly, even when the board is sparse.  
  
The architecture is secure and free from risk, with no apparent security risks. Separations of concerns should be followed, and relevant parameters should be configurable. If necessary, features should be switched if necessary.  
  
Testing includes unit tests, manual test plans, peer review testing, edge cases, invalid input validation, and input sanitization. The code has not been peer reviewed, and edge cases need explicit handling.  
  
In conclusion, the code meets the basic requirements for implementing a Connect4 game, but some features, such as the "Smart Computer (AI)" player, are not implemented. It is recommended to follow best practices, improve maintainability, and ensure proper testing and documentation.

**Game 4:**

The code implements classes and functionality for a word-based XO board game, meeting all stakeholder requirements. It is formatted correctly with consistent indentation and spacing, but some areas have extra whitespace. Best practices include single responsibility, error handling, and minimal nesting. The code has several magic values, but uses named constants for clarity. Comments are useful and not excessive.  
  
Maintainability is good, with meaningful class and function names and the DRY Principle. The Word\_XO class can be split into smaller components. Performance is acceptable for small game boards with simple operations. Optimization is needed to avoid using system("cls") in display\_board, which is expensive and platform dependent.  
  
Architecture is secure, with file operations verifying file validity and handling exceptions. Separation of concerns is followed, but Word\_XO could benefit from further modularization. Configurable parameters handle player types and board dimensions through user input and predefined logic. AI player types are switched dynamically.  
  
Testing includes unit tests for core methods, manual testing for edge cases, and input sanitization to avoid invalid states. Sufficient documentation is provided, including instructions on running the program, dependencies, and a basic overview of functionality.

**Game 5:**

The game implements numerical Tic-Tac-Toe with various player types (Human, Random AI, MiniMax AI), and the game mechanics seem functional. Stakeholder approval is required for implementation details, and the code is generally well-formatted. However, there are excessive blank lines in some sections and unnecessary whitespace. Best practices include adhering to the Single Responsibility Principle (SRP) and separating logic for AI behavior and move generation.  
  
MiniMax\_Player combines logic for both AI behavior and move generation, but it lacks error handling and logging mechanisms. It is recommended to define the constant 15 as a constexpr or named constant and avoid using raw integers for dimensions. Inline comments are sparse, and the code could benefit from more explanation.  
  
Maintainability includes refactoring into smaller methods, extracting large logic blocks into helper functions, consistent indentation, and color formatting. Repetition in is\_win and is\_draw and repeated board updates can also improve clarity. Method length should be broken into smaller functions for better clarity.  
  
Performance is acceptable, but the MiniMax algorithm lacks optimization and performance will degrade as board size or move complexity increases. Modern random libraries are preferred over rand() for move generation.  
  
Architecture includes security, separation of concerns, configurable parameters, and feature switches for debugging. Tests include unit tests, edge cases, manual tests, peer review, edge cases, input validation, and input sanitization.  
  
Insufficient documentation is provided, with minimal inline documentation and unclear purpose of methods like game\_is\_over or update\_board. A ReadMe.md file is recommended for further review and improvement.

**Code Review by Belal Gamal:**

**Game 2:**

**The code is functional and well-structured but can improve in the following areas:**

* **Use constants instead of magic values.**
* **Implement proper logging for errors.**
* **Add unit tests and test edge cases.**
* **Finalize AI implementation (replace commented code).**

**Game 3:**

* **Add Retry Mechanism for Invalid Inputs.**
* **Optimize Game-Over Logic**
* **Provide a placeholder for the AI player to avoid user confusion**
* **Comment the Code**

**Game5:**

* **Code Quality: Needs improvements in formatting, commenting, error handling, and adhering to best practices (e.g., SRP and DRY principle).**
* **Performance: AI logic needs optimization (e.g., alpha-beta pruning for the MiniMax algorithm).**
* **Architecture: The code should be refactored to follow the separation of concerns, with better configuration options and reduced reliance on magic values.**
* **Release Readiness: Missing release annotations and readiness checks.**

**Game6:**

* **Code Structure: Needs improvements in code separation, error handling, and configuration.**

**7. Performance: The AI logic requires optimization to handle larger game sizes or more complex logic.**

**8. Maintainability: Some functions are too long, repetitive, and lack proper documentation.**

**9. Testing: Lack of unit tests and manual testing.**

**10. Documentation: Missing comprehensive documentation for the game setup, play instructions, and system requirements.**

**11. Release Readiness: The game lacks versioning, release annotations, and a formal release plan.**

**Code Review by Mohamed Ehab:**

**Game1:**

**Have the requirements been met?**

The header file defines the necessary classes for a triangular Tic-Tac-Toe game, including the board, players, and random player. It seems to meet the basic requirements for setting up the game.

**Have stakeholder(s) approved the change?**

This information is not available in the code. Approval from stakeholders would typically be documented outside the codebase.

**Is the code formatted correctly?**

The code is generally well-formatted, with consistent indentation and spacing.

**Unnecessary whitespace removed?**

There doesn't appear to be unnecessary whitespace. The code is clean and easy to read.

**Follow Single Responsibility principle?**

Yes, the classes have clear responsibilities. Triangle\_XO\_Board manages the board, Triangle\_XO\_player represents a player, and Triangle\_XO\_Random\_player handles random moves.

**Are different errors handled correctly?**

Error handling is not explicitly shown in the provided code. It would be good to see how errors are managed, especially in game logic.

**Are errors and warnings logged?**

There is no logging in the provided code. Adding logging for errors and important events would be beneficial.

**Magic values avoided?**

The code does not seem to use magic values. Constants and enums could be used for better readability.

**No unnecessary comments?**

The comments are minimal and relevant. There are no unnecessary comments cluttering the code.

**Minimal nesting used?**

The code structure is flat and easy to follow, with minimal nesting.

Maintainability

**Is the code easy to read?**

Yes, the code is easy to read with clear class and method names.

**Is the code not repeated (DRY Principle)?**

The code does not appear to have unnecessary repetition. Common functionality is encapsulated in base classes.

**Is the code method/class not too long?**

The classes and methods are concise and focused on specific tasks.

Performance

**Is the code performance acceptable?**

The code uses efficient data structures like vector and pair. Performance should be acceptable for the intended use case.

**Is it secure/free from risk?**

The code does not handle security concerns explicitly, but it appears to be free from obvious risks.

**Are separations of concern followed?**

Yes, the code follows the separation of concerns principle, with different classes handling different aspects of the game.

**Relevant Parameters are configurable?**

The code does not show configurable parameters. Adding configuration options could improve flexibility.

**Feature switched if necessary?**

There is no mention of feature switches. This might not be necessary for this context.

**Do unit tests pass?**

There are no unit tests provided in the code. Adding unit tests would be essential for verifying functionality.

**Do manual test plans pass?**

This information is not available in the code. Manual testing should be documented separately.

**Has been peer review tested?**

This information is not available in the code. Peer reviews should be documented separately.

**Have edge cases been tested?**

Edge case handling is not shown in the code. It would be good to see how edge cases are managed.

**Are invalid inputs validated?**

Input validation is not explicitly shown. Adding validation would improve robustness.

**Are inputs sanitized?**

Input sanitization is not shown. This might be less critical for a game but is still a good practice.

**Is there sufficient documentation?**

The code has minimal comments. More detailed documentation would be helpful.

**Is the ReadMe.md file up to date?**

There is no ReadMe.md file provided. Adding one would be beneficial for understanding the project.

**Has the release been annotated (GA etc)?**

This information is not available in the code. Release annotations should be documented separately.

**Summary**

Overall, the code in Triangle\_XO.h is well-structured and follows good practices. It is functional, readable, and maintainable. However, there are areas for improvement, such as adding error handling, logging, input validation, and more detailed documentation. Additionally, implementing unit tests and ensuring stakeholder approval would be essential steps before finalizing the code.

**Game4:**

**Have the requirements been met?**

The header file defines the necessary classes for a word-based Tic-Tac-Toe game, including the board, players, and random player. It seems to meet the basic requirements for setting up the game.

**Have stakeholder(s) approved the change?**

This information is not available in the code. Approval from stakeholders would typically be documented outside the codebase.

**Is the code formatted correctly?**

The code is generally well-formatted, with consistent indentation and spacing.

**Unnecessary whitespace removed?**

There doesn't appear to be unnecessary whitespace. The code is clean and easy to read.

**Follow Single Responsibility principle?**

Yes, the classes have clear responsibilities. Word\_XO\_Board manages the board, Word\_XO\_player represents a player, and Word\_XO\_Random\_player handles random moves.

**Are different errors handled correctly?**

Error handling is present in some parts of the code, such as input validation in ReadNumber. However, more comprehensive error handling could be added.

**Are errors and warnings logged?**

There is no logging in the provided code. Adding logging for errors and important events would be beneficial.

**Magic values avoided?**

The code does not seem to use magic values. Constants and enums could be used for better readability.

**No unnecessary comments?**

The comments are minimal and relevant. There are no unnecessary comments cluttering the code.

**Minimal nesting used?**

The code structure is flat and easy to follow, with minimal nesting.

Maintainability

**Is the code easy to read?**

Yes, the code is easy to read with clear class and method names.

**Is the code not repeated (DRY Principle)?**

The code does not appear to have unnecessary repetition. Common functionality is encapsulated in base classes.

**Is the code method/class not too long?**

The classes and methods are concise and focused on specific tasks.

**Is the code performance acceptable?**

The code uses efficient data structures like vector, pair, and set. Performance should be acceptable for the intended use case.

**Is it secure/free from risk?**

The code does not handle security concerns explicitly, but it appears to be free from obvious risks.

**Are separations of concern followed?**

Yes, the code follows the separation of concerns principle, with different classes handling different aspects of the game.

**Relevant Parameters are configurable?**

The code does not show configurable parameters. Adding configuration options could improve flexibility.

**Feature switched if necessary?**

There is no mention of feature switches. This might not be necessary for this context.

**Do unit tests pass?**

There are no unit tests provided in the code. Adding unit tests would be essential for verifying functionality.

**Do manual test plans pass?**

This information is not available in the code. Manual testing should be documented separately.

**Has been peer review tested?**

This information is not available in the code. Peer reviews should be documented separately.

**Have edge cases been tested?**

Edge case handling is not shown in the code. It would be good to see how edge cases are managed.

**Are invalid inputs validated?**

Input validation is present in some parts of the code, such as ReadNumber. More comprehensive validation could be added.

**Are inputs sanitized?**

Input sanitization is not shown. This might be less critical for a game but is still a good practice.

**Is there sufficient documentation?**

The code has minimal comments. More detailed documentation would be helpful.

**Is the ReadMe.md file up to date?**

There is no ReadMe.md file provided. Adding one would be beneficial for understanding the project.

**Has the release been annotated (GA etc)?**

This information is not available in the code. Release annotations should be documented separately.

**Summary**

Overall, the code in Word\_XO.h is well-structured and follows good practices. It is functional, readable, and maintainable. However, there are areas for improvement, such as adding error handling, logging, input validation, and more detailed documentation. Additionally, implementing unit tests and ensuring stakeholder approval would be essential steps before finalizing the code.

**Game3:**

**Have the requirements been met?**

The header file defines the necessary classes and functions for a 5x5 Tic-Tac-Toe game, including the board, players, and random player. It seems to meet the basic requirements for setting up and playing the game.

**Have stakeholder(s) approved the change?**

This information is not available in the code. Approval from stakeholders would typically be documented outside the codebase.

**Is the code formatted correctly?**

The code is generally well-formatted, with consistent indentation and spacing.

**Unnecessary whitespace removed?**

There doesn't appear to be unnecessary whitespace. The code is clean and easy to read.

**Follow Single Responsibility principle?**

Yes, the classes have clear responsibilities. TicTacToe5x5 manages the board, tic\_tac\_Player represents a player, and tic\_tac\_Random\_Player handles random moves.

**Are different errors handled correctly?**

Error handling is present in some parts of the code, such as input validation in getmove. However, more comprehensive error handling could be added.

**Are errors and warnings logged?**

There is no logging in the provided code. Adding logging for errors and important events would be beneficial.

**Magic values avoided?**

The code does not seem to use magic values. Constants and enums could be used for better readability.

**No unnecessary comments?**

The comments are minimal and relevant. There are no unnecessary comments cluttering the code.

**Minimal nesting used?**

The code structure is flat and easy to follow, with minimal nesting.

**Is the code easy to read?**

Yes, the code is easy to read with clear class and method names.

**Is the code not repeated (DRY Principle)?**

The code does not appear to have unnecessary repetition. Common functionality is encapsulated in base classes.

**Is the code method/class not too long?**

The classes and methods are concise and focused on specific tasks.

**Is the code performance acceptable?**

The code uses efficient data structures like vector and pair. Performance should be acceptable for the intended use case.

**Is it secure/free from risk?**

The code does not handle security concerns explicitly, but it appears to be free from obvious risks.

**Are separations of concern followed?**

Yes, the code follows the separation of concerns principle, with different classes handling different aspects of the game.

**Relevant Parameters are configurable?**

The code does not show configurable parameters. Adding configuration options could improve flexibility.

**Feature switched if necessary?**

There is no mention of feature switches. This might not be necessary for this context.

**Do unit tests pass?**

There are no unit tests provided in the code. Adding unit tests would be essential for verifying functionality.

**Do manual test plans pass?**

This information is not available in the code. Manual testing should be documented separately.

**Has been peer review tested?**

This information is not available in the code. Peer reviews should be documented separately.

**Have edge cases been tested?**

Edge case handling is not shown in the code. It would be good to see how edge cases are managed.

**Are invalid inputs validated?**

Input validation is present in some parts of the code, such as getmove. More comprehensive validation could be added.

**Are inputs sanitized?**

Input sanitization is not shown. This might be less critical for a game but is still a good practice.

**Is there sufficient documentation?**

The code has minimal comments. More detailed documentation would be helpful.

**Is the ReadMe.md file up to date?**

There is no ReadMe.md file provided. Adding one would be beneficial for understanding the project.

**Has the release been annotated (GA etc)?**

This information is not available in the code. Release annotations should be documented separately.

**Summary**

Overall, the code in TicTacToe5x5.h is well-structured and follows good practices. It is functional, readable, and maintainable. However, there are areas for improvement, such as adding error handling, logging, input validation, and more detailed documentation. Additionally, implementing unit tests and ensuring stakeholder approval would be essential steps before finalizing the code.

**Game6:**

**Have the requirements been met?**

The header file defines the necessary classes and functions for a Misere Tic-Tac-Toe game, including the board, players, and random player. It seems to meet the basic requirements for setting up and playing the game.

**Have stakeholder(s) approved the change?**

This information is not available in the code. Approval from stakeholders would typically be documented outside the codebase.

Code Formatting

**Is the code formatted correctly?**

The code is generally well-formatted, with consistent indentation and spacing.

**Unnecessary whitespace removed?**

There doesn't appear to be unnecessary whitespace. The code is clean and easy to read.

**Follow Single Responsibility principle?**

Yes, the classes have clear responsibilities. MisereTicTacToeBoard manages the board, MisereTicTacToePlayer represents a player, and MisereTicTacToeRandomPlayer handles random moves.

**Are different errors handled correctly?**

Error handling is present in some parts of the code, such as input validation in getmove. However, more comprehensive error handling could be added.

**Are errors and warnings logged?**

There is no logging in the provided code. Adding logging for errors and important events would be beneficial.

**Magic values avoided?**

The code does not seem to use magic values. Constants and enums could be used for better readability.

**No unnecessary comments?**

The comments are minimal and relevant. There are no unnecessary comments cluttering the code.

**Minimal nesting used?**

The code structure is flat and easy to follow, with minimal nesting.

**Is the code easy to read?**

Yes, the code is easy to read with clear class and method names.

**Is the code not repeated (DRY Principle)?**

The code does not appear to have unnecessary repetition. Common functionality is encapsulated in base classes.

**Is the code method/class not too long?**

The classes and methods are concise and focused on specific tasks.

**Is the code performance acceptable?**

The code uses efficient data structures like vector and pair. Performance should be acceptable for the intended use case.

**Is it secure/free from risk?**

The code does not handle security concerns explicitly, but it appears to be free from obvious risks.

**Are separations of concern followed?**

Yes, the code follows the separation of concerns principle, with different classes handling different aspects of the game.

**Relevant Parameters are configurable?**

The code does not show configurable parameters. Adding configuration options could improve flexibility.

**Feature switched if necessary?**

There is no mention of feature switches. This might not be necessary for this context.

**Do unit tests pass?**

There are no unit tests provided in the code. Adding unit tests would be essential for verifying functionality.

**Do manual test plans pass?**

This information is not available in the code. Manual testing should be documented separately.

**Has been peer review tested?**

This information is not available in the code. Peer reviews should be documented separately.

**Have edge cases been tested?**

Edge case handling is not shown in the code. It would be good to see how edge cases are managed.

**Are invalid inputs validated?**

Input validation is present in some parts of the code, such as getmove. More comprehensive validation could be added.

**Are inputs sanitized?**

Input sanitization is not shown. This might be less critical for a game but is still a good practice.

**Is there sufficient documentation?**

The code has minimal comments. More detailed documentation would be helpful.

**Is the ReadMe.md file up to date?**

There is no ReadMe.md file provided. Adding one would be beneficial for understanding the project.

**Has the release been annotated (GA etc)?**

This information is not available in the code. Release annotations should be documented separately.

**Summary**

Overall, the code in MisereTicTacToe.h is well-structured and follows good practices. It is functional, readable, and maintainable. However, there are areas for improvement, such as adding error handling, logging, input validation, and more detailed documentation. Additionally, implementing unit tests and ensuring stakeholder approval would be essential steps before finalizing the code.

***GitHub Private Repo Picture:***

