**Faculty of Computers and Artificial Intelligence**

**Cairo University**

**OOP**

**Dr. Mohamed EL\_Ramly**

**Assignment 2 Task 2,3,4,5**

|  |  |  |  |
| --- | --- | --- | --- |
| Name: | IDS: | Emails: | Section: |
| Marwan Tamer Sayed Ali | 20230380 | marwantamer004@gmail.com | S19 |
| Mohamed Refaat Mohamed Awd | 20230337 | mrefaat8853@gmail.com | S19 |
| Rana Tarek Ahmed Fouad Ibrahim | 20230632 | r.tarekahmed@gmail.com | S20 |

|  |  |
| --- | --- |
| Name: | Tasks: |
| Marwan Tamer Sayed Ali | Game 2, 5 \ Main function |
| Mohamed Refaat Mohamed Awd | Game 1, 4, 8 |
| Rana Tarek Ahmed Fouad Ibrahim | Game 3, 6, 7 |

**1. Pyramid Game**

**PyramidBoard** (Template Class):

Description: Represents a pyramid-shaped game board, inherited from the Board class. The board has 3 rows with increasing columns in each row (1, 3, 5).

Attributes:

* rows and columns define the board dimensions.
* board is a 2D array that stores the game state (characters).
* n\_moves keeps track of the number of moves made.

Methods:

* update\_board(int x, int y, T symbol): Updates the board with a symbol at position (x, y).
* display\_board(): Displays the board in a pyramid format.
* is\_win(): Checks for a winning condition (horizontal, vertical, or diagonal).
* is\_draw(): Checks if the game is a draw (no winner and all cells filled).
* game\_is\_over(): Returns true if the game is either won or drawn.

**X\_O\_Player** (Template Class):

* **Description**: Represents a human player in the Pyramid game.
* **Attributes**: Inherits from Player<T>, storing the player's name and symbol.
* **Methods**:
  + getmove(int& x, int& y): Prompts the user to enter the coordinates for the next move.

**X\_O\_Random\_Player** (Template Class):

* **Description**: Represents an AI player that selects random moves.
* **Attributes**: Inherits from RandomPlayer<T>, stores the player's symbol and name.
* **Methods**:
  + getmove(int& x, int& y): Randomly selects an available position on the board.

**Suggested Improvements:**

**UI Enhancement**: Improve the user interface by making the board display clearer. Currently, it's hard to see where the columns and rows are. Consider using symbols or a more intuitive format

2. **Four in a Row Game** (Connect Four)

 **FourInARowBoard** (Template Class):

* **Description**: Represents the game board for the "Four In a Row" game, inheriting from the Board class. The board has 6 rows and 7 columns, and the objective is to connect four symbols in a row.
* **Attributes**:
  + rows and columns define the board dimensions.
  + board is a 2D array representing the state of each cell ('.' for empty cells).
  + n\_moves keeps track of the number of moves made.
* **Methods**:
  + get\_first\_empty\_row(int col): Returns the first empty row in the given column.
  + update\_board(int x, int y, T symbol): Updates the board with a symbol in the first available row of the specified column.
  + display\_board(): Displays the board.
  + is\_win(): Checks if there's a winner by checking horizontal, vertical, and diagonal lines.
  + is\_draw(): Checks if the game is a draw (no winner and all cells filled).
  + game\_is\_over(): Returns true if the game is either won or drawn.
* **Friend Class**: FourInARowMinimaxPlayer<T> is declared as a friend, meaning it can access private and protected members of this class.

 **FourInRowPlayer** (Template Class):

* **Description**: Represents a human player in the "Four In a Row" game.
* **Attributes**: Inherits from Player<T>, stores the player's name and symbol.
* **Methods**:
  + getmove(int& x, int& y): Prompts the player to enter a column number where they want to drop their symbol. The row is determined by the first available space in that column.

**Suggested Improvements:**

* **Column Full Validation:** Add a more intuitive method of notifying the player when a column is full. This can be done with a message after a player attempts an invalid move.
* **Move Preview**: Let the player see a preview of where their move will land before confirming the move, which enhances usability and reduces user frustration**.**

**3.X\_O\_Board5X5** **Game**

**1. X\_O\_Board5X5:**

This class represents the game board for a 5x5 grid-based game of Tic-Tac-Toe (X and O). It inherits from the Board<T> class and provides methods to:

* **update\_board(int x, int y, T symbol)**: Updates the board at position (x, y) with the given symbol if the position is empty.
* **display\_board()**: Displays the current state of the board.
* **is\_win()**: Checks if there is a winner by counting three-in-a-row combinations for each player ('X' and 'O').
* **is\_draw()**: Checks if the game is a draw (if the number of three-in-a-row combinations for both players is the same and all moves are made).
* **game\_is\_over()**: Returns true if the game is over (either a win or draw), otherwise false.
* **count\_threes(T symbol)**: Helper function that counts the number of three-in-a-row combinations for a given symbol (either 'X' or 'O'). It checks rows, columns, and diagonals.

**2. X\_O\_HumanPlayer:**

This class represents a human player in the game. It inherits from the Player<T> class and provides a method to get the player's move:

* **getmove(int& x, int& y)**: Prompts the player to enter a row and column for their move.

**3. X\_O\_RandomPlayer:**

This class represents a random AI player. It inherits from the RandomPlayer<T> class and provides a method to generate a random move:

* **getmove(int& x, int& y)**: Generates a random move by selecting a random row and column.

**Suggested Improvements:**

**Improvement:** The display could be simplified and made more user-friendly

**5.** **Numerical Tic-Tac-Toe Game**

**NumericalBoard Class:**

* This class defines the board for the game. It has methods for updating the board (update\_board), displaying the board (display\_board), checking for a win (is\_win), checking for a draw (is\_draw), and determining if the game is over (game\_is\_over).
* The check\_sum method is used to determine if three numbers sum to 15, which is the winning condition (similar to the magic square rule).
* The update\_board method checks if the move is valid (in bounds, not already occupied) and updates the board with the player's symbol.

**HumanPlayer Class:**

* This class represents a human player in the game. The getmove method allows the player to select a number and make a move (choosing an empty cell on the board).
* The available numbers are displayed for the player, and the player can choose a valid number that hasn't been used already.
* The method validates the move and updates the board with the selected number, ensuring the move is legal and the number hasn't been used globally**.**

**NumericalRandomPlayer Class:**

* This class represents a computer player that selects random valid moves from the available numbers and positions.
* It follows a similar approach as the human player but without player input, randomly selecting an available move.

**Global State:**

* globalUsedPositions tracks all used positions on the board.
* globalUsedNumbers tracks all numbers that have been used by any player.

**Suggested Improvements:**

**Display Enhancements**: The display board method currently prints the board in a readable way. Consider adding more features like highlighting the winning combination**.**



**6. X\_O\_Board\_misere Game**

**X\_O\_Board\_misere Class**

* **Constructor:** Initializes a 3x3 board with all positions set to 0 (empty).
* **update\_board(int x, int y, T mark):** Updates the board with the player's mark at the specified position. If the player chooses to undo the move (mark = 0), it decrements the move count and clears the board position.
* **display\_board():** Displays the current state of the board, showing the coordinates and marks in the cells.
* **is\_win():** Checks whether there's a winning combination of three marks in a row, column, or diagonal.
* **is\_draw():** Returns true if all positions are filled and there's no winner**.**
* **game\_is\_over():** Returns true if the game is over, either due to a win or a draw**.**

**X\_O\_Player\_misere Class**

* **Constructor:** Initializes a player with a name and a mark (symbol).
* **getmove(int& x, int& y):** Prompts the player to enter the coordinates for their next move**.**

**X\_O\_Random\_Player\_misere Class**

* **Constructor:** Initializes a random player, sets the dimension of the board to 3, and gives the player the name "Random Computer Player".
* **getmove(int& x, int& y):** Randomly selects coordinates between 0 and 2 for the next move**.**

**Suggested Improvements:**

**--**



**7.** **4x4 Tic-Tac-Toe Game:**

**Board Class (X\_O\_Board<T>):**

* Represents a 4x4 grid.
* Initializes with some pre-filled tokens (X and O) at specific positions.
* Enforces rules for token movement, including adjacency and ownership validation.
* Supports win, draw, and game-over checks.
* Offers a display function to print the board's current state**.**

**Human Player Class (X\_O\_Human\_Player<T>):**

* Allows a human player to select a token and move it to a valid adjacent position.
* Validates input thoroughly to ensure moves comply with the game's rules**.**

**Random Player Class (X\_O\_Game\_Random\_Player<T>):**

* Implements a random AI player that moves a token to a valid adjacent position randomly**.**

**Template Design:**

* Makes the game generic and adaptable to different types of tokens (char, int, etc.).

**Rules:**

* Movement is limited to adjacent cells, as checked by is\_adjacent().
* Each player can only move their own tokens.
* Moves are restricted to empty cells**.**

**Win Conditions:**

* The code checks for 3 consecutive identical symbols horizontally, vertically, or diagonally.
* The win logic assumes a traditional Tic-Tac-Toe-like pattern, adjusted for the 4x4 grid.

**Draw Conditions:**

* A draw occurs when all cells are filled without a win condition being met.

**Human Player Input:**

* Input is robustly validated, ensuring the chosen token belongs to the player and the move is within valid bounds**.**

**Random Player Logic:**

* Randomly picks a token and destination, ensuring they follow adjacency and empty cell rules.

**Suggestions for Improvement:**

Initialization: The X\_O\_Board constructor hardcodes initial tokens (X and O). This design assumes a specific starting layout, which may limit flexibility for different setups.



**8. Ultimate Tic Tac Toe Game:**

**Board Class (Small\_XO\_Board<T>)**

* Represents a 3x3 grid for a single Tic Tac Toe board.
* Initialization:
  + Creates an empty 3x3 board.
  + Tracks the winning symbol (X or O) when a win occurs.
* Rules:
  + Supports token placement with update\_board(), ensuring moves are within bounds.
  + Enforces win conditions (3 in a row horizontally, vertically, or diagonally).
  + Checks for a draw when all cells are filled without a winner.
* Game Status:
  + Provides methods (is\_win(), is\_draw(), game\_is\_over()) to validate the current game state**.**
* Display:
  + Offers display\_board() to show the current board layout with symbols or indices for emptycells.

**Ultimate Board Class (Ultimate\_XO\_Board<T>)**

* Represents a 3x3 grid of Small\_XO\_Board instances, forming a 9x9 ultimate Tic Tac Toe game.
* Initialization:
  + Dynamically allocates and initializes 9 Small\_XO\_Board objects.
* Rules:
  + Token placement is mapped to the appropriate small board, ensuring valid moves.
  + Win conditions are checked for the 3x3 grid of small boards, based on which player wins the majority of boards.
  + Draws occur when all small boards are either won or drawn, and no overall winner exists.
* Game Status:
  + Combines statuses from small boards to determine the overall game state (win, draw, or ongoing).
* Display:
  + Provides a comprehensive view of the entire game state, including individual small boards.

**Human Player Class (Ultimate\_Player<T>)**

* Allows a human player to make moves**.**
* Rules:
  + Validates input to ensure the selected cell belongs to the current player and is within bounds.
* Interaction:
  + Guides the user to select a valid move, displaying error messages for invalid input.

**Random Player Class (Ultimate\_Random\_Player<T>)**

* Implements a basic AI that selects random valid moves.
* **Rules:**
  + Ensures the move adheres to adjacency, ownership, and empty cell rules**.**
* **Behavior:**
  + Randomly selects a valid cell and destination, offering a simple yet functional AI experience.

**Template Design**

* Generic Implementation:
  + Uses templates (<T>) to adapt the game for various token types (e.g., char, int).



**UML:**

