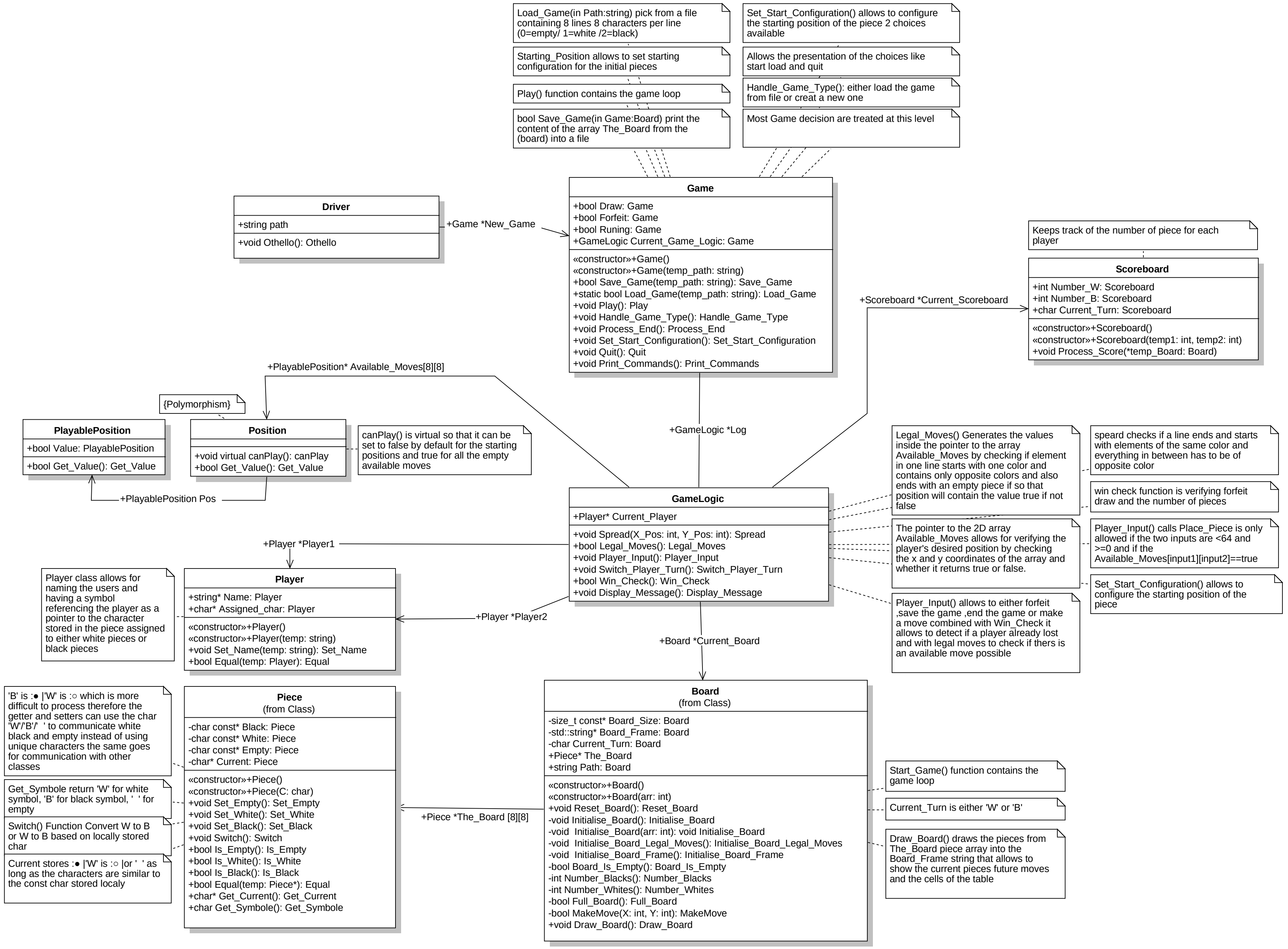
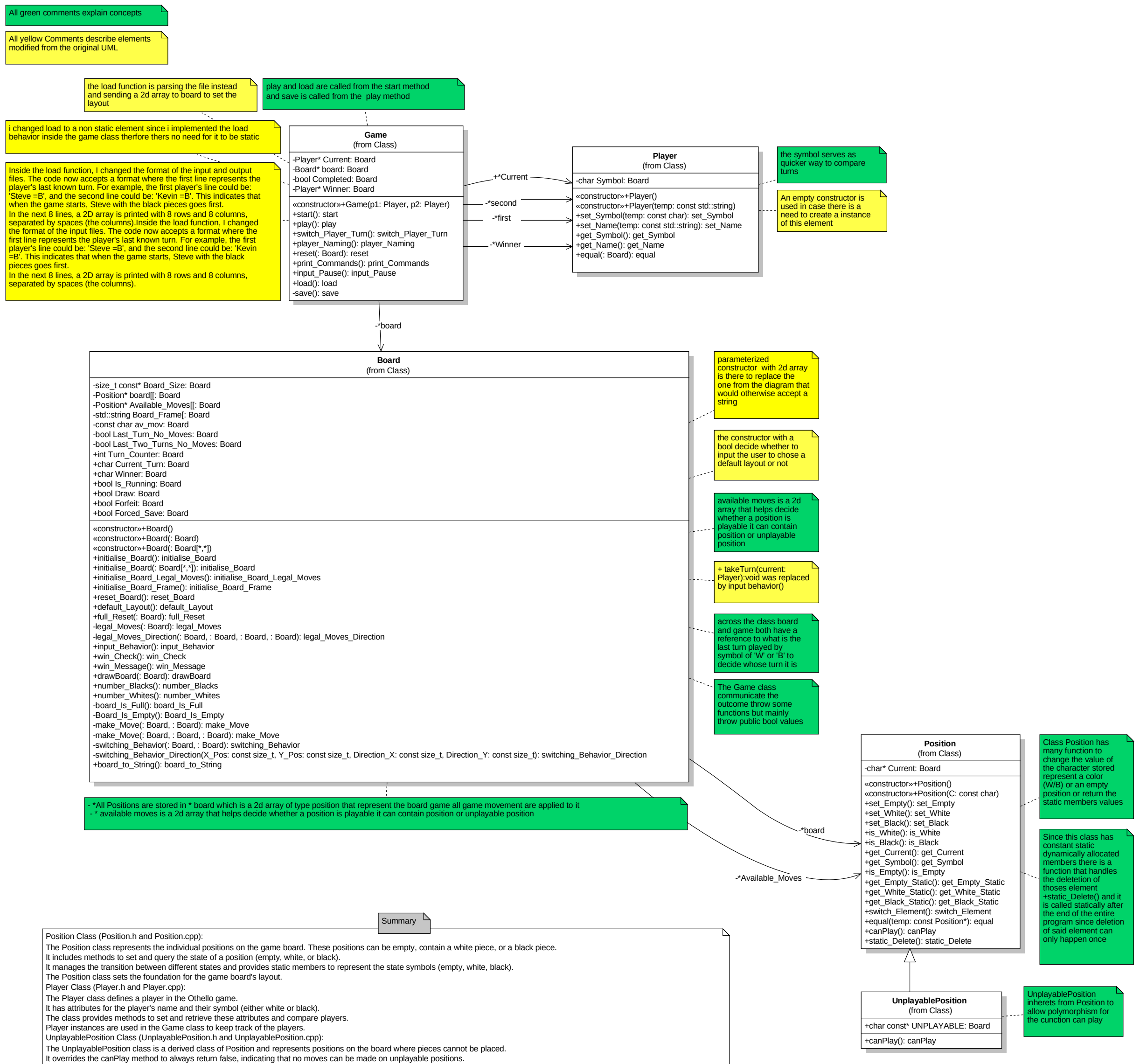

Load_Game(in Path:string) pick from a file containing 8 lines 8 characters per line (0=empty/ 1=white /2=black)
Starting_Position allows to set starting configuration for the initial pieces
Play() function contains the game loop
bool Save_Game(in Game:Board) print the content of the array The_Board from the (board) into a file
Set_Start_Configuration() allows to configure the starting position of the piece 2 choices available
Allows the presentation of the choices like start load and quit
Handle_Game_Type(): either load the game from file or creat a new one
Most Game decision are treated at this level
Driver
+string path
+void Othello(): Othello
+Game *New_Game
Game
+bool Draw: Game
+bool Forfeit: Game
+bool Runing: Game
+GameLogic Current_Game_Logic: Game
«constructor»+Game()
«constructor»+Game(temp_path: string)
+bool Save_Game(temp_path: string): Save_Game
+static bool Load_Game(temp_path: string): Load_Game
+void Play(): Play
+void Handle_Game_Type(): Handle_Game_Type
+void Process_End(): Process_End
+void Set_Start_Configuration(): Set_Start_Configuration
+void Quit(): Quit
+void Print_Commands(): Print_Commands
Keeps track of the number of piece for each player
Scoreboard
+int Number_W: Scoreboard
+int Number_B: Scoreboard
+char Current_Turn: Scoreboard
«constructor»+Scoreboard()
«constructor»+Scoreboard(temp1: int, temp2: int)
+void Process_Score(*temp_Board: Board)
+Scoreboard *Current_Scoreboard
+PlayablePosition* Available_Moves[8][8]
{Polymorphism}
PlayablePosition
+bool Value: PlayablePosition
+bool Get_Value(): Get_Value
Position
+void virtual canPlay(): canPlay
+bool Get_Value(): Get_Value
canPlay() is virtual so that it can be set to false by default for the starting positions and true for all the empty available moves
+PlayablePosition Pos
+GameLogic *Log
GameLogic
+Player* Current_Player
+void Spread(X_Pos: int, Y_Pos: int): Spread
+bool Legal_Moves(): Legal_Moves
+void Player_Input(): Player_Input
+void Switch_Player_Turn(): Switch_Player_Turn
+bool Win_Check(): Win_Check
+void Display_Message(): Display_Message
Legal_Moves() Generates the values inside the pointer to the array Available_Moves by checking if element in one line starts with one color and contains only opposite colors and also ends with an empty piece if so that position will contain the value true if not false
speard checks if a line ends and starts with elements of the same color and everything in between has to be of opposite color
win check function is verifying forfeit draw and the number of pieces
The pointer to the 2D array Available_Moves allows for verifying the player's desired position by checking the x and y coordinates of the array and whether it returns true or false.
Player_Input() calls Place_Piece is only allowed if the two inputs are <64 and >=0 and if the Available_Moves[input1][input2]==true
Set_Start_Configuration() allows to configure the starting position of the piece
Player_Input() allows to either forfeit ,save the game ,end the game or make a move combined with Win_Check it allows to detect if a player already lost and with legal moves to check if thers is an available move possible
+Player *Player1
+Player *Player2
Player class allows for naming the users and having a symbol referencing the player as a pointer to the character stored in the piece assigned to either white pieces or black pieces
Player
+string* Name: Player
+char* Assigned_char: Player
«constructor»+Player()
«constructor»+Player(temp: string)
+void Set_Name(temp: string): Set_Name
+bool Equal(temp: Player): Equal
+Board *Current_Board
'B' is :● |'W' is :○ which is more difficult to process therefore the getter and setters can use the char 'W'/'B'/' ' to communicate white black and empty instead of using unique characters the same goes for communication with other classes
Get_Symbole return 'W' for white symbol, 'B' for black symbol, ' ' for empty
Switch() Function Convert W to B or W to B based on locally stored char
Current stores :● |'W' is :○ |or ' ' as long as the characters are similar to the const char stored localy
Piece
(from Class)
-char const* Black: Piece
-char const* White: Piece
-char const* Empty: Piece
-char* Current: Piece
«constructor»+Piece()
«constructor»+Piece(C: char)
+void Set_Empty(): Set_Empty
+void Set_White(): Set_White
+void Set_Black(): Set_Black
+void Switch(): Switch
+bool Is_Empty(): Is_Empty
+bool Is_White(): Is_White
+bool Is_Black(): Is_Black
+bool Equal(temp: Piece*): Equal
+char* Get_Current(): Get_Current
+char Get_Symbole(): Get_Symbole
+Piece *The_Board [8][8]
Board
(from Class)
-size_t const* Board_Size: Board
-std::string* Board_Frame: Board
-char Current_Turn: Board
+Piece* The_Board
+string Path: Board
«constructor»+Board()
«constructor»+Board(arr: int)
+void Reset_Board(): Reset_Board
-void Initialise_Board(): Initialise_Board
-void Initialise_Board(arr: int): void Initialise_Board
-void Initialise_Board_Legal_Moves(): Initialise_Board_Legal_Moves
-void Initialise_Board_Frame(): Initialise_Board_Frame
-bool Board_Is_Empty(): Board_Is_Empty
-int Number_Blacks(): Number_Blacks
-int Number_Whites(): Number_Whites
-bool Full_Board(): Full_Board
-bool MakeMove(X: int, Y: int): MakeMove
+void Draw_Board(): Draw_Board
Start_Game() function contains the game loop
Current_Turn is either 'W' or 'B'
Draw_Board() draws the pieces from The_Board piece array into the Board_Frame string that allows to show the current pieces future moves and the cells of the table


Game
(from Class)
-Player* Current: Board
-Board* board: Board
-bool Completed: Board
-Player* Winner: Board
«constructor»+Game(p1: Player, p2: Player)
+start(): start
+play(): play
+switch_Player_Turn(): switch_Player_Turn
+player_Naming(): player_Naming
+reset(: Board): reset
+print_Commands(): print_Commands
+input_Pause(): input_Pause
+load(): load
-save(): save
+*Current
-*second
-*first
-*Winner
Player
(from Class)
-char Symbol: Board
«constructor»+Player()
«constructor»+Player(temp: const std::string)
+set_Symbol(temp: const char): set_Symbol
+set_Name(temp: const std::string): set_Name
+get_Symbol(): get_Symbol
+get_Name(): get_Name
+equal(: Board): equal
-*board
Board
(from Class)
-size_t const* Board_Size: Board
-Position* board[[: Board
-Position* Available_Moves[[: Board
-std::string Board_Frame[: Board
-const char av_mov: Board
-bool Last_Turn_No_Moves: Board
-bool Last_Two_Turns_No_Moves: Board
+int Turn_Counter: Board
+char Current_Turn: Board
+char Winner: Board
+bool Is_Running: Board
+bool Draw: Board
+bool Forfeit: Board
+bool Forced_Save: Board
«constructor»+Board()
«constructor»+Board(: Board)
«constructor»+Board(: Board[*,*])
+initialise_Board(): initialise_Board
+initialise_Board(: Board[*,*]): initialise_Board
+initialise_Board_Legal_Moves(): initialise_Board_Legal_Moves
+initialise_Board_Frame(): initialise_Board_Frame
+reset_Board(): reset_Board
+default_Layout(): default_Layout
+full_Reset(: Board): full_Reset
-legal_Moves(: Board): legal_Moves
-legal_Moves_Direction(: Board, : Board, : Board, : Board): legal_Moves_Direction
+input_Behavior(): input_Behavior
+win_Check(): win_Check
+win_Message(): win_Message
+drawBoard(: Board): drawBoard
+number_Blacks(): number_Blacks
+number_Whites(): number_Whites
-board_Is_Full(): board_Is_Full
-Board_Is_Empty(): Board_Is_Empty
-make_Move(: Board, : Board): make_Move
-make_Move(: Board, : Board, : Board): make_Move
-switching_Behavior(: Board, : Board): switching_Behavior
-switching_Behavior_Direction(X_Pos: const size_t, Y_Pos: const size_t, Direction_X: const size_t, Direction_Y: const size_t): switching_Behavior_Direction
+board_to_String(): board_to_String
-*board
-*Available_Moves
Position
(from Class)
-char* Current: Board
«constructor»+Position()
«constructor»+Position(C: const char)
+set_Empty(): set_Empty
+set_White(): set_White
+set_Black(): set_Black
+is_White(): is_White
+is_Black(): is_Black
+get_Current(): get_Current
+get_Symbol(): get_Symbol
+is_Empty(): is_Empty
+get_Empty_Static(): get_Empty_Static
+get_White_Static(): get_White_Static
+get_Black_Static(): get_Black_Static
+switch_Element(): switch_Element
+equal(temp: const Position*): equal
+canPlay(): canPlay
+static_Delete(): static_Delete
UnplayablePosition
(from Class)
+char const* UNPLAYABLE: Board
+canPlay(): canPlay

All green comments explain concepts

All yellow Comments describe elements modified from the original UML

the load function is parsing the file instead and sending a 2d array to board to set the layout

play and load are called from the start method and save is called from the play method

i changed load to a non static element since i implemented the load behavior inside the game class therfore thers no need for it to be static

Inside the load function, I changed the format of the input and output files. The code now accepts a format where the first line represents the player's last known turn. For example, the first player's line could be: 'Steve =B', and the second line could be: 'Kevin =B'. This indicates that when the game starts, Steve with the black pieces goes first.
In the next 8 lines, a 2D array is printed with 8 rows and 8 columns, separated by spaces (the columns).Inside the load function, I changed the format of the input files. The code now accepts a format where the first line represents the player's last known turn. For example, the first player's line could be: 'Steve =B', and the second line could be: 'Kevin =B'. This indicates that when the game starts, Steve with the black pieces goes first.
In the next 8 lines, a 2D array is printed with 8 rows and 8 columns, separated by spaces (the columns).

the symbol serves as quicker way to compare turns

An empty constructor is used in case there is a need to create a instance of this element

parameterized constructor with 2d array is there to replace the one from the diagram that would otherwise accept a string

the constructor with a bool decide whether to input the user to chose a default layout or not

available moves is a 2d array that helps decide whether a position is playable it can contain position or unplayable position

+ takeTurn(current: Player):void was replaced by input behavior()

across the class board and game both have a reference to what is the last turn played by symbol of 'W' or 'B' to decide whose turn it is

The Game class communicate the outcome throw some functions but mainly throw public bool values

- *All Positions are stored in * board which is a 2d array of type position that represent the board game all game movement are applied to it
- * available moves is a 2d array that helps decide whether a position is playable it can contain position or unplayable position

Class Position has many function to change the value of the character stored represent a color (W/B) or an empty position or return the static members values

Since this class has constant static dynamically allocated members there is a function that handles the deletetion of thoses element +static_Delete() and it is called statically after the end of the entire program since deletion of said element can only happen once

UnplayablePosition inherets from Position to allow polymorphism for the cunction can play

Summary

Position Class (Position.h and Position.cpp):
The Position class represents the individual positions on the game board. These positions can be empty, contain a white piece, or a black piece.
It includes methods to set and query the state of a position (empty, white, or black).
It manages the transition between different states and provides static members to represent the state symbols (empty, white, black).
The Position class sets the foundation for the game board's layout.
Player Class (Player.h and Player.cpp):
The Player class defines a player in the Othello game.
It has attributes for the player's name and their symbol (either white or black).
The class provides methods to set and retrieve these attributes and compare players.
Player instances are used in the Game class to keep track of the players.
UnplayablePosition Class (UnplayablePosition.h and UnplayablePosition.cpp):
The UnplayablePosition class is a derived class of Position and represents positions on the board where pieces cannot be placed.
It overrides the canPlay method to always return false, indicating that no moves can be made on unplayable positions.
Unplayable positions are used to represent the fixed pieces at the center of the Othello board that cannot be changed during the game.
Board Class (Board.h and Board.cpp):
The Board class represents the game board and is composed of Position objects.
It contains methods to initialize the board, display the current state, check for valid moves, and manage the game's status.
The class includes methods for resetting the board, calculating the number of pieces, and generating messages for the game's outcome.
The board state is updated based on player moves.
Game Class (Game.h and Game.cpp):
The Game class manages the core game logic, including player turns, command input, and game flow.
It interacts with Player objects to keep track of players' names and symbols.
The class has methods to start and play the game, switch player turns, and handle loading and saving game states.
The game is initiated and controlled through the start method, which includes the main game loop and command processing.
File Input/Output (Load and Save):
The Game class provides methods for loading and saving the game state from/to text files. These methods interact with the file system to read and write the game board's layout and the current player's turn.
Interactions in the Main Program (Assignment 3.cpp):
In the main program, two players are created, and their names are checked for equality.
If the player names are different, a dynamic instance of the Game class is created with these players.
The game is initiated using the start method, which handles game commands, player input, and the main game loop.
After the game concludes, the dynamic Game instance is deleted.
The program also ensures that any dynamically allocated static members within the Position class are deleted after the game.
These components work together to simulate the Othello (Reverse) game, allowing players to interact with the game board, make moves, and manage the game's state, including the player names and scores. The game can also be saved and loaded from text files, providing a comprehensive gaming experience. The main program orchestrates the entire game simulation from start to finish.