SECJ1023 PROGRAMMING TECHINIQUE II

20242025 – SEMESTER 1

DESIGN ANALYSIS

FACULTY OF MJIIT

GROUP A&C

NAME	MATRIC ID
Kahlan Sultan Mohammed	A23MJ4021
Liu Ruoyang	A23MJ4022
Bu Guoshun	A23MJ4019
Abdulrahman Siad	A23MJ3061
Hassan Saad	A23MJ3005

Brainstorm Meeting Video link:

https://www.youtube.com/watch?v=3O0-2Ghnguo

Requirement Analysis

This project aims to develop an **E-Connect 4** game using the C++ programming language and its Object-Oriented Programming (OOP) features. The WinBGIm graphics library is utilized to create a graphical user interface, providing visual elements, mouse interactions, and other functionalities. Below is the detailed requirement analysis:

1. Functional Requirements

Game Rules Implementation:

The game should implement standard Connect 4 rules, including board initialization, turn-based gameplay, and victory detection.

Gameplay Modes:

Support two modes: Player vs Player (PvP) and Player vs Computer (PvC).

In PvC mode, an AI opponent should be able to make logical decisions based on game states.

Graphical User Interface (GUI):

Utilize the WinBGIm library to:

Display a 6x7 board layout graphically.

Show player information, such as current player turn and victory messages.

Enable interactive buttons like "Start Game," "Play Again," and "Exit Game."

Allow players to make moves by clicking on the desired column using the mouse.

2. Non-Functional Requirements

Scalability:

Design the code structure with OOP principles (e.g., encapsulation, inheritance, polymorphism) to allow for future extensions, such as adding AI difficulty levels.

Maintainability:

Ensure a modular code structure with clear separation of functionalities (e.g., MainMenu, GameScreen) for easier debugging and maintenance.

Performance:

Guarantee a smooth user experience by minimizing graphical flickering and ensuring responsive UI interactions.

3. Technical Requirements

Programming Language: Use C++ with an emphasis on OOP features, such as classes, objects, inheritance, and polymorphism.

Graphics Library: WinBGIm, to load and show graphical elements, process mouse input, and handle game interface components.

Development Environment: A compatible IDE that supports the WinBGIm graphics library. We chose Visual Studio Code due to its strong ability in extension and modularization.

4. User Experience Requirements

Provide a user-friendly interface where players can interact easily without memorizing complex commands.

Ensure that graphical elements are visually clear, with buttons and messages positioned appropriately.

Class and Object Analysis

This section explains the **objects**, **classes**, and **relationships** in the E-Connect 4 project. The design uses Object-Oriented Programming (OOP) concepts, making it clear and easy to extend or modify.

1. Objects

The main objects in this project are:

Player: A person or AI playing the game.
Button: A clickable area in the game (e.g., "Play Again," "Exit Game").
Disc: A piece dropped on the board during gameplay.
Board: The grid where the game is played.
GameState: Keeps track of whether the game is ongoing, won, or drawn.

2. Classes and Their Functions

Here are the classes in the project, along with their functions and purposes:

a. **Player** (Base Class):

Purpose: Represents a general player in the game.
Functions:
Player(const char* name, int id): Initializes the player with a name and ID (1 for Player 1, 2 for Player 2).
displayTurn(): Displays whose turn it is. For example, "Player 1's Turn".

b. **HumanPlayer** (Inherits from Player):

Purpose: Represents a human player who interacts with the game using a mouse.
Functions:
HumanPlayer(const char* name, int id): Creates a human player with a name and ID.
displayTurn(): Displays a message for the human player's turn (e.g., "Player 1's Turn").

c. **AIPlayer** (Inherits from Player):

Purpose: Represents the computer player in PvC mode.
Functions:

AIPlayer(const char* name, int id): Creates an AI player with a name and ID.
getMove(int board[6][7]): Decides the best move based on the current state of the board.
checkVictory(int board[6][7], int playerID): Checks if a given player can win in the next move.
isColumnPlayable(int board[6][7], int col): Checks if a column is valid for dropping a disc.

d. **Button**:

Purpose: Represents a button in the game, such as "Exit Game" or "Play Again".

Functions:
Button(int x, int y, int width, int height, const char* label): Creates a button with a position, size, and label.
draw(): Draws the button on the screen.
isClicked(int mouseX, int mouseY): Checks if the button was clicked by the mouse.

e. **GameScreen**:

Purpose: Manages the gameplay interface and logic.

Functions:
GameScreen(bool isPvC): Creates the game screen and decides whether the game is PvP or PvC.
display(): Draws the game board, player info, and other interface elements.
handleInput(int x, int y): Handles player actions based on mouse input.
highlightWinningDiscs(): Highlights the four connected discs when someone wins.
drawBoard(): Draws the Connect 4 board with current disc positions.
checkVictory(): Checks if any player has won.
displayVictory(): Shows the victory screen after a player wins.
resetGame(): Resets the board and game state for a new match.

f. **MainMenu**:

Purpose: Represents the game's main menu.

Functions:
MainMenu(): Creates the main menu with all its buttons.
display(): Draws the main menu screen with title and buttons.
handleClick(int mouseX, int mouseY): Detects which button the player clicked.

g. **HelpScreen**:

Purpose: Displays the rules or instructions for the game.

Functions:
HelpScreen(): Creates the help screen with a back button.
display(): Draws the help screen with instructions.
handleBack(int x, int y): Checks if the back button is clicked to return to the main menu.

3. Class Relationships

a. MainMenu and Button

Relationship: Composition

Reason:

The MainMenu class contains an array of Button objects, representing menu buttons like "Exit Game" and "PvP".

These Button objects are fully managed by MainMenu, as they are instantiated directly within its constructor and are part of its lifetime. If the MainMenu object is destroyed, the Button objects cease to exist.

b. HelpScreen and Button

Relationship: Composition

Reason:

The HelpScreen class contains a "Back" button.

The HelpScreen class is responsible for creating and managing the backButton object, and its lifetime is tied to that of the HelpScreen object.

c. GameScreen and Button

Relationship: Aggregation

Reason:

The GameScreen class uses Button objects temporarily in its displayVictory method. These objects are declared as local variables and are used only for rendering purposes.

The GameScreen class does not manage the lifetime of these Button objects beyond the scope of the method where they are used.

d. GameScreen and Player

Relationship: Composition

Reason:

The GameScreen class contains a Player array, representing the two players in the game. These players are instantiated in GameScreen's constructor and destroyed in its destructor.

GameScreen fully manages the lifecycle of the Player objects, and they cannot exist independently of the GameScreen object.

e. Player and its Subclasses (HumanPlayer and AIPlayer)

Relationship: Inheritance

Reason:

HumanPlayer and AIPlayer inherit from the Player base class.

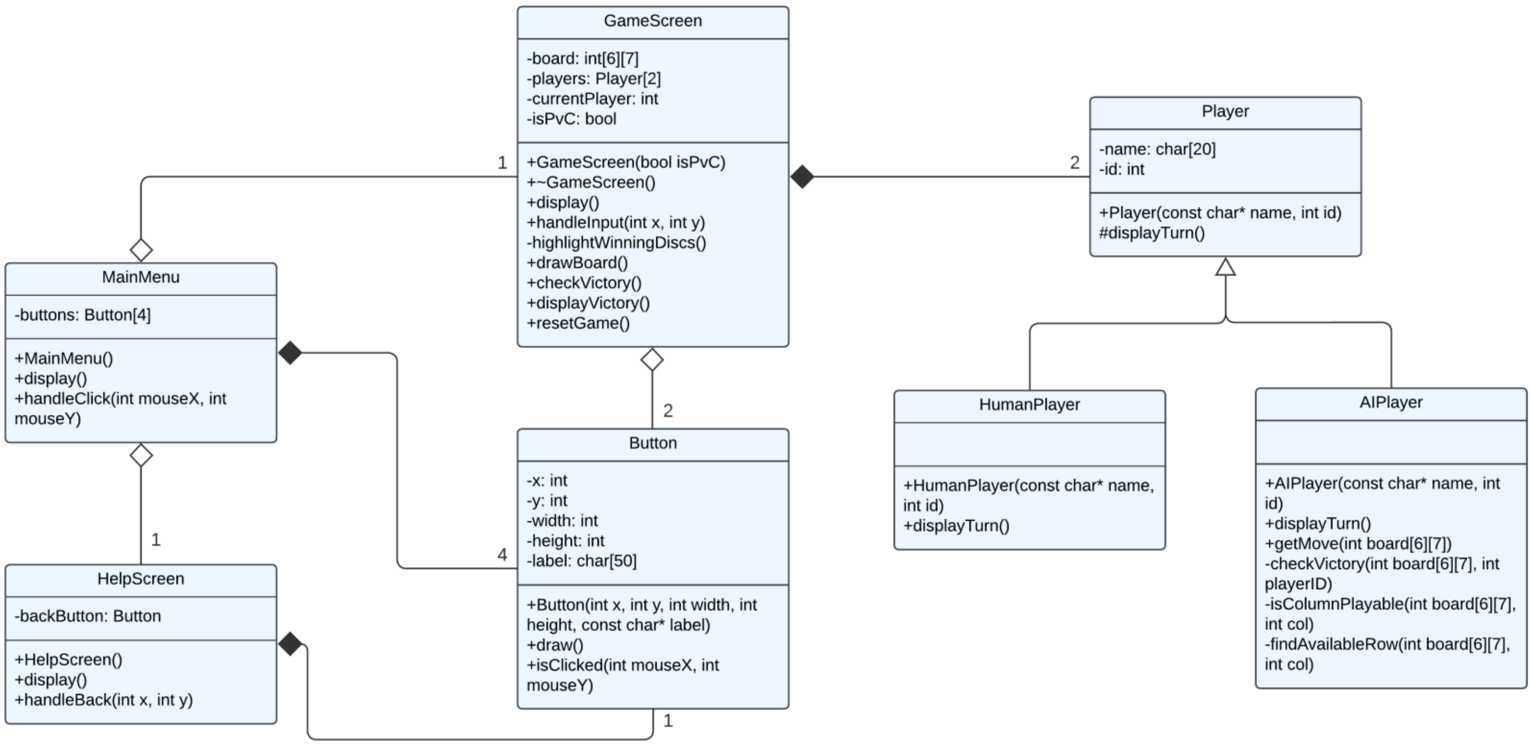
They override specific methods, such as displayTurn and provide unique implementations for player behavior.

g. MainMenu and GameScreen

Relationship: Aggregation
Reason:
The MainMenu class interacts with GameScreen through the state mechanism in the main program logic, but it does not instantiate or directly manage GameScreen objects.

h. MainMenu and HelpScreen
Relationship: Aggregation
Reason:
The MainMenu class interacts with HelpScreen through the state mechanism in the main program logic, but it does not instantiate or directly manage HelpScreen objects.

4. Class diagram



This class diagram shows the relationships between classes.