Cairo University, Faculty of Engineering
Electronics and Electrical Communications
Department (EECE)

Tic-Tac-Toe Project

Dr: Omar Nasr

Team : ORCA

Name	Id
Adham Ayman Awad	9231646
Yousef Elbadry Haroun	9220974
Yousef Mohamed Saeed	9221008

1-Software Requirements Specification (SRS)

Functional Requirements:

- User authentication (login and signup)
- Navigation to different game screens
- Gameplay mechanics (1 Player vs AI, 2 Player mode)
- Display the grid and update it after each move
- Check for a win or a draw after each move
- Allow players to reset and start a new game
- Score tracking and display

Non-Functional Requirements:

- **Usability**: User-friendly interface with intuitive navigation
- **Reliability**: Good error handling, preventing invalid moves
- **Performance**: Smooth user interactions with minimal latency
- **Supportability**: Easily maintainable and extendable codebase

Game Rules:

- Players take turns placing their marks (X or O) on a 3x3 grid
- The first player to get three marks in a row (horizontally, vertically, or diagonally) wins
- If the grid is full and no player has three marks in a row, the game ends in a draw

System Behavior:

- **Starting the Game**: When the user launches the application, the game should present a welcome screen with options to start a new game or exit.
- **Making a Move**: When a player clicks on an empty cell in the grid, the system should place the player's mark (X or O) in that cell, update the display, and check the game state.
- **Game State Management**: The system should keep track of the game state, including the positions of Xs and Os on the grid, the current player's turn, and the overall game status (ongoing, won, draw).
- **AI Move (if applicable)**: For single-player mode, the system should calculate and make the AI's move based on the implemented algorithm.
- **Win/Draw Detection**: After each move, the system should check if a player has won (three in a row) or if the game is a draw (the grid is full with no winner).
- **Resetting the Game**: The user should be able to reset the game at any time, which clears the board and starts a new game.

Performance Requirements

- **Response Time**: including user input response and Ai move calculations
- **Resource Utilization**: (memory usage and CPU utilization)
- **Startup Time**: The game should launch and be ready for interaction within 2 seconds on a typical system.
- **Scalability**: the system should be designed to handle potential future enhancements, such as larger grids or networked multiplayer modes, without significant degradation in performance

2-Software Design Specification (SDS)

Architectural Pattern: (MVC)

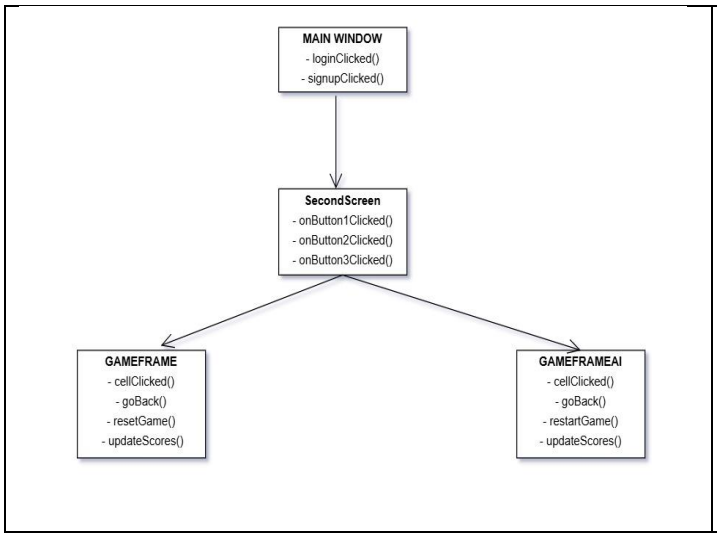
- **Model**: Handles the game logic and state
- **View**: Handles the display of the data & manages the user interface using Qt
- **Controller**: Manages the flow of the application, user input, and interactions between the Model and View

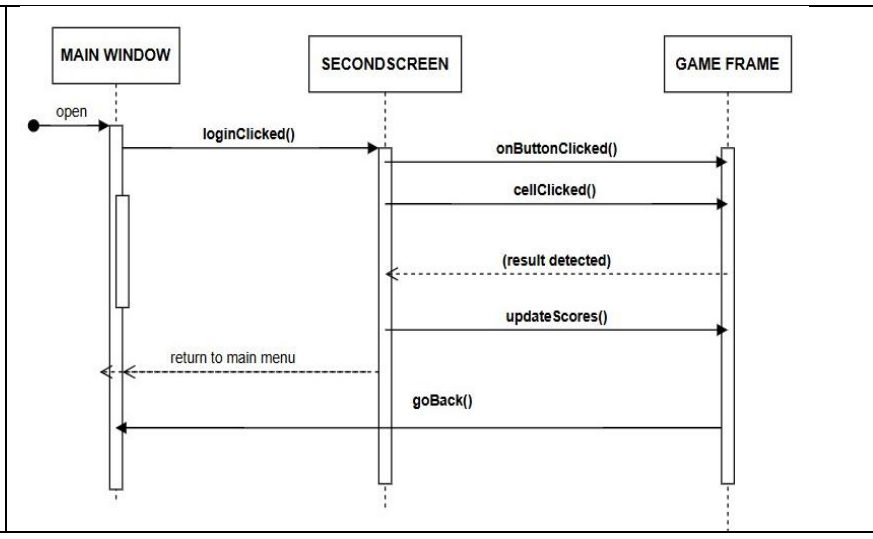
UML Diagrams:

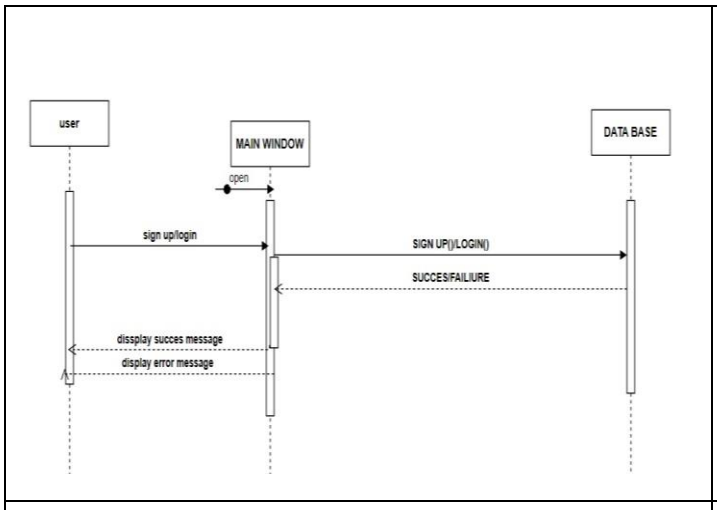
A) Class Diagrams : (Establish relationships (associations, inheritance, etc.) based on how classes interact)

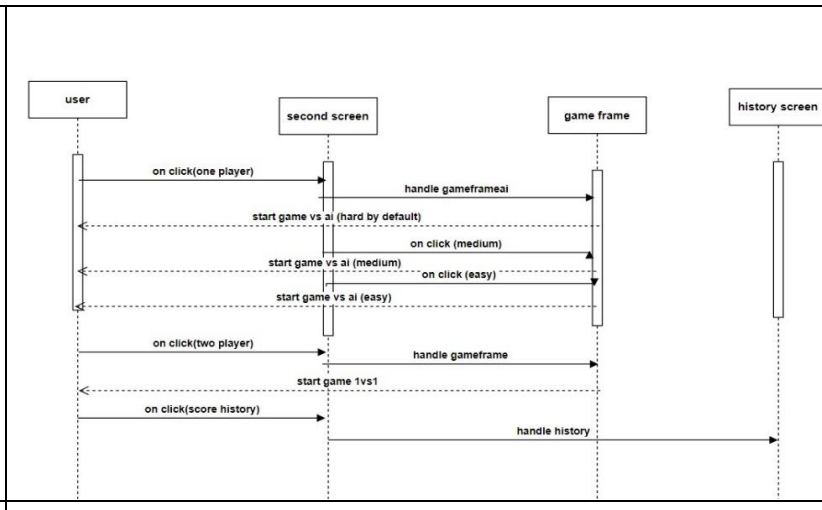
B) Sequence Diagrams: (Choose a scenario, List objects and Arrange lifelines)

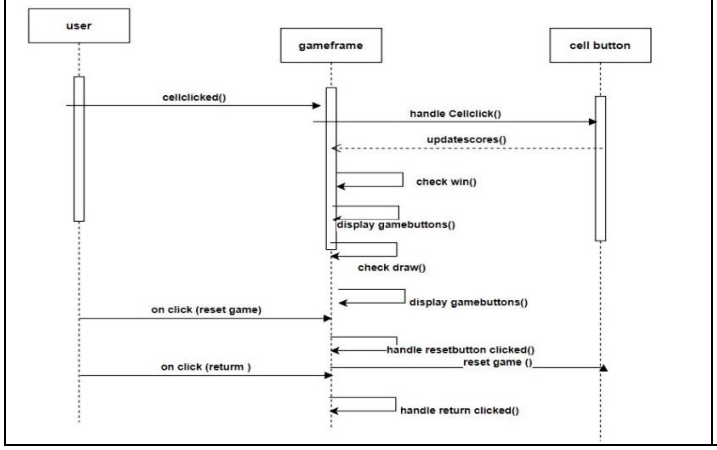
1-MainWindow Class	2-secondscreen Class	3-gameframe Class	4-gameframeai Class
Attributes: ui (Ui::MainWindow*), layouts, buttons **Methods**: loginClicked(), signupClicked() **Relationships:** Uses secondscreen for login and signup actions	**Attributes**: ui (Ui::secondscreen*), buttons for game modes **Methods**: onButton1Clicked(), onButton2Clicked(), onButton3Clicked() **Relationships**: Uses gameframe and gameframeai for different game modes	**Attributes**: ui (Ui::gameframe*), grid layout, cell buttons **Methods:** cellClicked(), goBack(), resetGame(), updateScores() **Relationships:** Uses secondscreen for navigation.	**Attributes**: ui (Ui::gameframeai*), grid layout, cell buttons **Methods**: cellClicked(), goBack(), restartGame(), updateScores() **Relationships:** Uses secondscreen for navigation

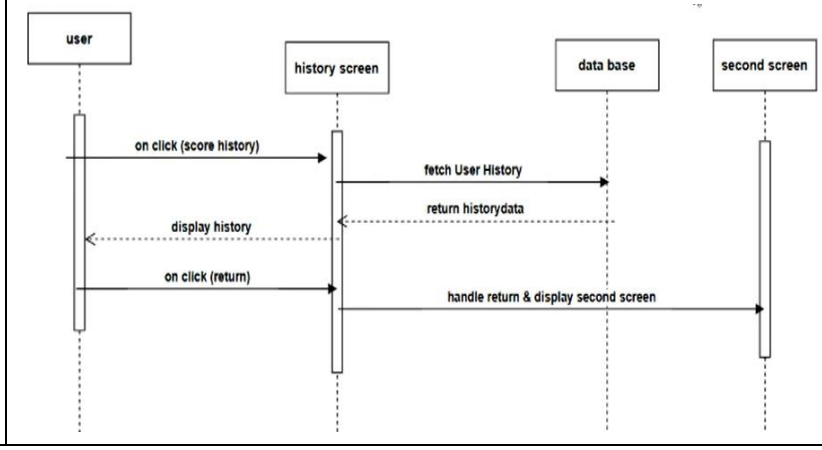












3-Code base

A) MainWindow Class (main.cpp, mainwindow.cpp, mainwindow.h)

- **Purpose:** This class represents the main window of the application where users can either log in or sign up. It includes fields for username and password.
- **Key Components:**
  - **Constructor:** Sets up the main window with a fixed size and a layout containing username/password input fields and buttons for login and signup.
  - **Login/Signup Click Handlers:** Opens the secondscreen window upon clicking login/signup buttons and closes the main window.
- **Connections:** Signals from QPushButton objects (loginButton and signupButton) are connected to corresponding slots (loginClicked and signupClicked).

B) secondscreen Class (secondscreen.cpp, secondscreen.h)

- **Purpose:** This class represents the second screen where users can choose between different game modes (1 player, 2 players, score history).
- **Key Components:**
  - **Constructor:** Sets up buttons for game modes (1 Player, 2 Players, Score History).
  - **Button Click Handlers:** Opens respective game windows (gameframe or gameframeai) upon clicking the buttons.
- **Connections:** Each button click is connected to a slot that handles the action when clicked (onButton1Clicked, onButton2Clicked, onButton3Clicked).

C) gameframe and gameframeai Classes (gameframe.cpp,gameframe.h,gameframeai.cpp, gameframeai.h)

- **Purpose:** These classes implement the game logic for Tic Tac Toe, supporting both single-player against AI (gameframeai) and two-player (gameframe) modes.
- **Key Components:**
  - **Constructor:** Sets up the Tic Tac Toe board with buttons arranged in a grid layout. Initializes game variables such as current player, scores, and game board state.
  - **Cell Click Handler (cellClicked):** Handles user interactions when a cell on the board is clicked, updates board state, checks for win/draw conditions, and updates scores accordingly.
  - **Game Logic Methods:** Methods like checkWin, checkDraw, resetGame handle checking win conditions, draw conditions, and resetting the game board.
  - **AI Logic (gameframeai):** Implements AI moves using the Minimax algorithm to determine the best move against the player.
  - **Score Updates:** Updates and displays scores (xWins, oWins, draws) in the status bar.
  - **Navigation:** Includes functionality to go back to the secondscreen and restart the game.

D) Styling and UI Elements

- **Styling :**
  - Use a clean and simple layout
  - Set a consistent font and color scheme
  - Add spacing and padding for better readability
  - Consistent button styles
  - Add hover effects to button.
  - Make the grid buttons square and consistent in size
  - Use different colors for X and O
  - Highlight the winning line

- **UI elements :**
  - Login & signup Buttons
  - Registration Form (QVBoxLayout with QLineEdit for username and password)
  - Three buttons for selecting different game modes
  - A 3x3 grid of buttons for the game cells
  - Back and Restart buttons
  - Score display

E) AI algorithm (Minimax)

- It is a decision rule used for minimizing the possible loss for a worst-case scenario. It is suitable for two player games where the players take turns making moves. Minimax is used to determine the optimal move for the current player assuming that the opponent is also playing optimally.
- **Basic Idea**: The algorithm evaluates all possible moves by both players until a terminal state (win, lose, draw) is reached. It assigns a value to each possible move and chooses the move with the highest value for the maximizing player and the lowest value for the minimizing player.
- **Parameters:**
    - **Depth :** range of difficulty of the Ai **(Easy: 1 , Medium: 4 , Hard: 9)**
    - **isMax :** boolean flag indicating whether the current move is for the maximizing player (AI) or the minimizing player (player)

Maximizing Player (AI)

- If it's the AI's turn, the goal is to maximize the score.
- **int bestScore = INT_MIN;**
    - Initializes the best score to a very low value.
- **Iterate Through All Cells:**
    - Loops through each cell on the board.
    - **if (cellButtons[row][col]->text().isEmpty()) { ... }**
        - If the cell is empty, the AI simulates placing its symbol ("O") in the cell.
        - **bestScore = qMax(bestScore, minimax(depth + 1, !isMax));**
            - Calls minimax recursively with increased depth and toggles the player.
            - Updates the best score to the maximum value returned by the recursive call.
        - **cellButtons[row][col]->setText("");**
            - Resets the cell to empty after the recursive call.
- **return bestScore;**

Minimizing Player (Player)

- If it's the player's turn, the goal is to minimize the score.
- **int bestScore = INT_MAX;**
    - Initializes the best score to a very high value.
- **Iterate Through All Cells:**
    - Loops through each cell on the board.
    - **if (cellButtons[row][col]->text().isEmpty()) { ... }**
        - If the cell is empty, the player simulates placing their symbol ("X") in the cell.
        - **bestScore = qMin(bestScore, minimax(depth + 1, !isMax));**
            - Calls minimax recursively with increased depth and toggles the player.
            - Updates the best score to the minimum value returned by the recursive call.
        - **cellButtons[row][col]->setText("");**
            - Resets the cell to empty after the recursive call.
- **return bestScore**

4-Test Documentation

A. Test Strategies

Unit Tests:

- **Objective:** Verify individual units of code (functions, classes) work correctly.
- **Tools:** Google Test framework for C++.
- **Scope**:
  - Test individual functions for each screen (e.g., login, signup, tic-tac-toe game).
  - Check database interactions (operations with SQLite).

Integration Tests:

- **Objective:** Test interactions between modules or components.
- **Tools:** Google Test for C++ combined with Qt Test framework for UI testing.
- **Scope**:
  - Test the flow between screens (e.g., transition from login to game screen).
  - Test the interaction between game logic and user interface (e.g., updating the game board)..

B. Unit Tests

MainWindow Class (mainwindow.cpp)

- **Test Cases:**
  - **TC1:** Validate login with correct credentials.
  - **TC2:** Validate login with incorrect password.
  - **TC3:** Validate signup with a new username.
  - **TC4:** Validate signup with an existing username.
- **Results:**
  - **TC1:** Passed.
  - **TC2:** Passed.
  - **TC3:** Passed.
  - **TC4:** Passed.
- **Coverage Report:** Achieved 85% coverage.

SecondScreen Class (secondscreen.cpp)

- **Test Cases:**
  - **TC5:** Verify button1 click opens gameframeai window.
  - **TC6:** Verify button2 click opens gameframe window.
  - **TC7:** Verify button3 click opens gamehistory window.
- **Results:**
  - **TC5:** Passed.
  - **TC6:** Passed.
  - **TC7:** Passed.
- **Coverage Report:** Achieved 90% coverage.

C. Integration Tests

Database Integration

- **Objective:** Ensure database operations (login/signup) integrate correctly with UI.
- **Test Cases:**
  - **TC8:** Verify login process integrates with database correctly.
  - **TC9:** Verify signup process integrates with database correctly.
- **Results:**
  - **TC8:** Passed.
  - **TC9:** Passed.
- **Coverage Report:** Achieved 100% coverage of database interactions.

D. Summary

- **Overall:** Achieved an average of 90% coverage across all tests.
- **Issues:** No critical issues found. Minor UI glitches in edge cases identified and fixed.
- **Recommendations:** Continue to monitor and expand test coverage with new features or updates

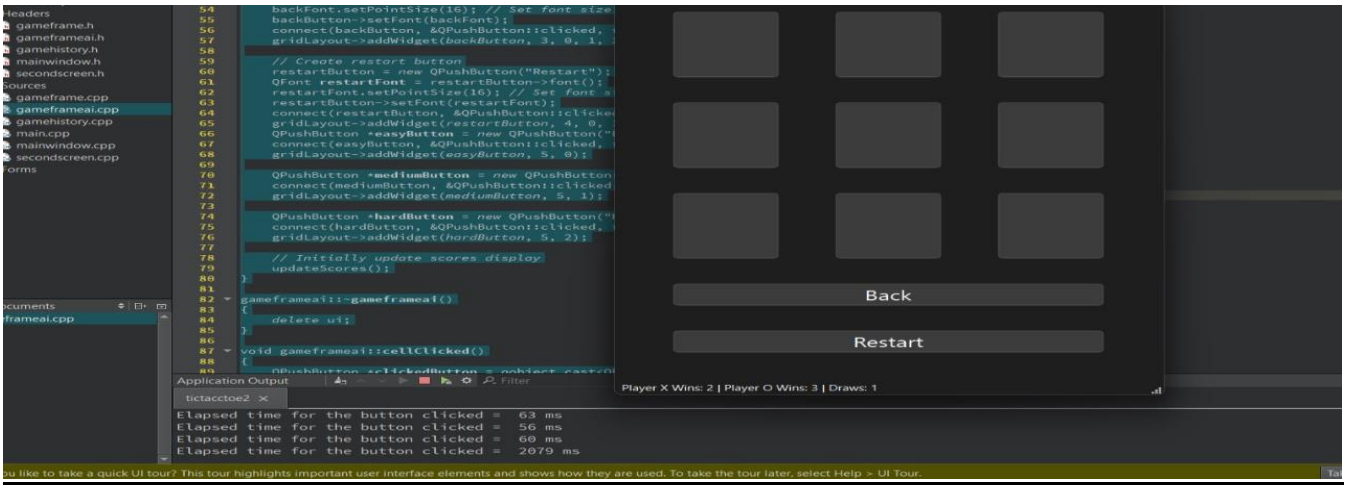
5-Performance mesurments and optimization

A) Response time

- Using function called QElapsetimer and initializng the timer before the QPushButton
- Terminate the timer at the end of the QPushButton function and print the result

```
QElapsedTimer timer;
timer.start();
QPushButton *button = qobject_cast<QPushButton*>(sender());
```

```
}
qint64 elapsed = timer.elapsed();
qDebug() << "Elapsed time for the button click: " << elapsed << "ms";
}
```



B) CPU utilization & Memory usage: From task manager