Department of Electronics and

Communication Engineering

Cairo University
Faculty of Engineering
Electronics and Communications Engineering Department

Data Structure
Advanced Tic Tac Toe Game

Submitted by:

Name	Section	ID
Ahmed Mostafa Mohamed Mohamedy	1	9230187
Ahmed Abdelrahman abdelrazeq Mohamed	1	9230152
Ahmed mansour amin mohamed	1	9230188
Hadey Sayed Mohamed Saber	4	9230972
Islam Hamdy Mohamed Mostafa	1	9230215

Table Of Contents

1.Introduction

Tic Tac Toe is a simple two-player game traditionally played on a 3x3 grid. The objective is to place three of your marks in a horizontal, vertical, or diagonal row. While the game is often used as a learning exercise for basic programming, this project elevates it into a fully featured software application. It includes user authentication, persistent data storage, a smart AI opponent using decision-making algorithms, and a polished graphical interface.

This project is designed as part of the Data Structures course to apply core principles of software engineering, object-oriented design, and data management in a real-world context.

2.Objectives

- Design and implement an interactive Tic Tac Toe game using C++.
- Incorporate a user-friendly GUI using Qt or a similar library.
- Develop a secure login and registration system with hashed passwords.
- Store and manage individual game histories for each user.
- Integrate a challenging AI opponent using the minimax algorithm.
- Write comprehensive unit and integration tests using Google Test.
- Set up a CI/CD pipeline using GitHub Actions.
- Optimize the application for performance, including CPU and memory usage

3.System Architecture

The system consists of the following major components:

- **GUI Layer**: User interaction through the graphical interface.
- **Game Engine**: Core logic for managing gameplay.
- **AI Module**: Decision-making logic for computer opponent.
- **User Management**: Handles login, registration, and session tracking.
- **Database/Storage Layer**: Maintains user profiles and game histories.

4.Software Requirements Specification (SRS)

4.1 Functional Requirements

- **Gameplay Mechanics**:
  - ★ Users can play in Player vs Player or Player vs AI modes.
  - ★ Game board updates dynamically with each move.
  - ★ System detects wins, losses, and draws.
- **User Management**:
  - ★ Users can register and log in securely.
  - ★ Passwords are stored securely using hashing.
  - ★ Session management keeps users logged in during use.
- **Game History**:
  - ★ Game results are saved per user.
  - ★ Players can view past game summaries.
  - ★ Option to replay past games.
- **AI Functionality**:
  - ★ The system can simulate a smart opponent.
  - ★ AI calculates moves using the minimax algorithm.

4.2 Non-Functional Requirements

- **Performance**:
  - ★ Application must respond to player actions within 200ms.
  - ★ Efficient memory and CPU usage.
- **Usability**:
  - ★ GUI must be intuitive and accessible.

    - ★ Clear feedback provided after every action.
- **Security**:
    - ★ All user credentials stored using secure hashing algorithms.
    - ★ Input validation to prevent SQL/file injection or code misuse.
- **Portability**:
    - ★ Application should run on Windows and Linux platforms.
- **Maintainability**:
    - ★ Code should follow the Google C++ Style Guide.
    - ★ Modular design for easier debugging and updates.

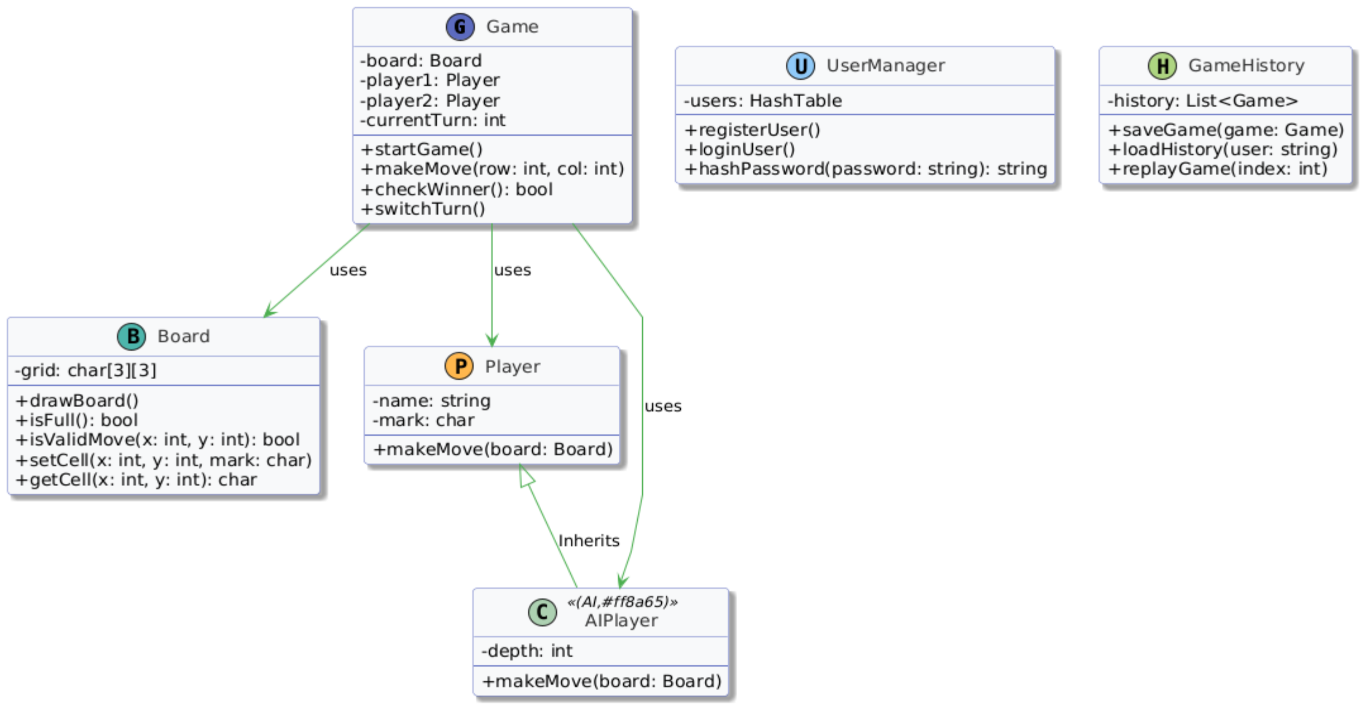
5.Software Design Specification (SDS)

This section describes the software architecture and the structural design of the Tic Tac Toe system. It includes the class structure, component interactions, use cases, and logic flows.

5.1 Class Design

The application follows an object-oriented design. Key classes include:

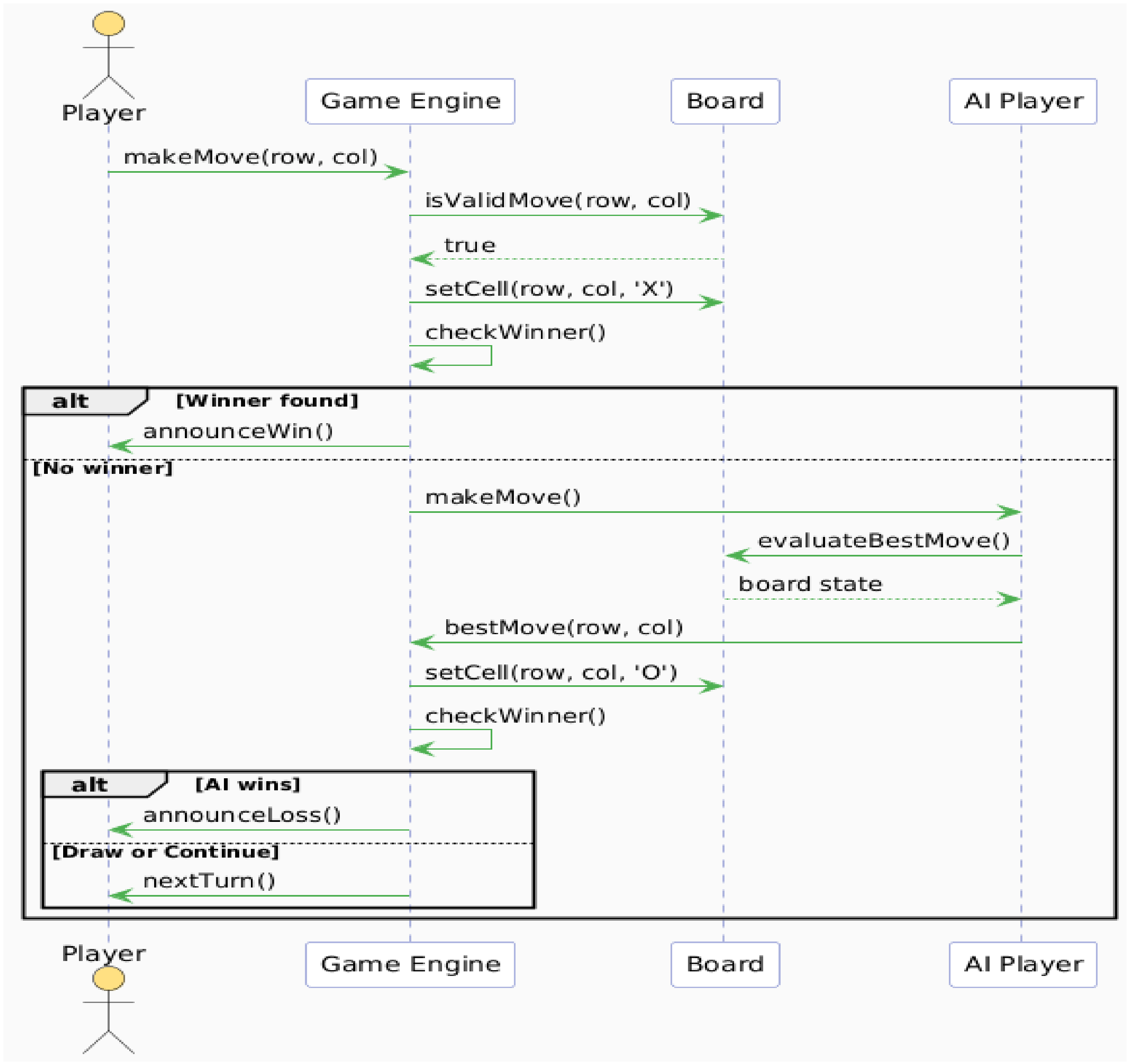
- **Game**: Manages the overall game logic.
- **Board**: Represents the 3x3 grid and board operations.
- **Player**: Base class for human and AI players.
- **AIPlayer**: Inherits from Player and implements decision logic.
- **UserManager**: Handles user registration and login.
- **Game History**: Manages storage and replay of past games.



.

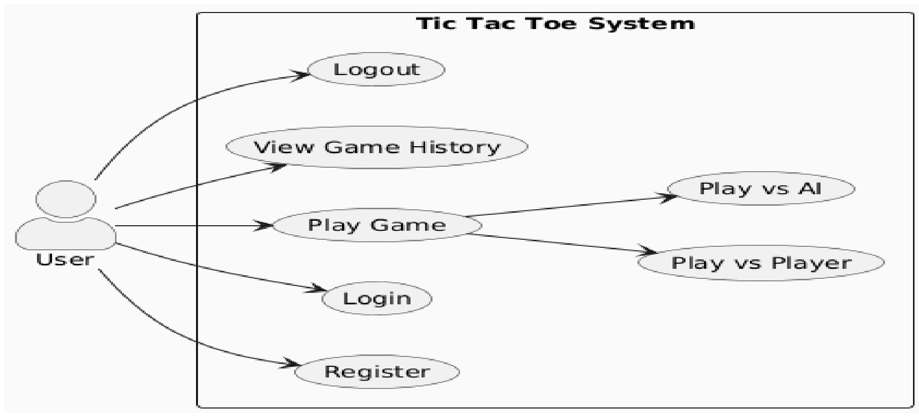
5.2 Sequence of Operations

The flow of control for a player making a move is illustrated in the sequence diagram. It demonstrates interaction between **Player**, **Game**, **Board**, and **AIPlayer**.



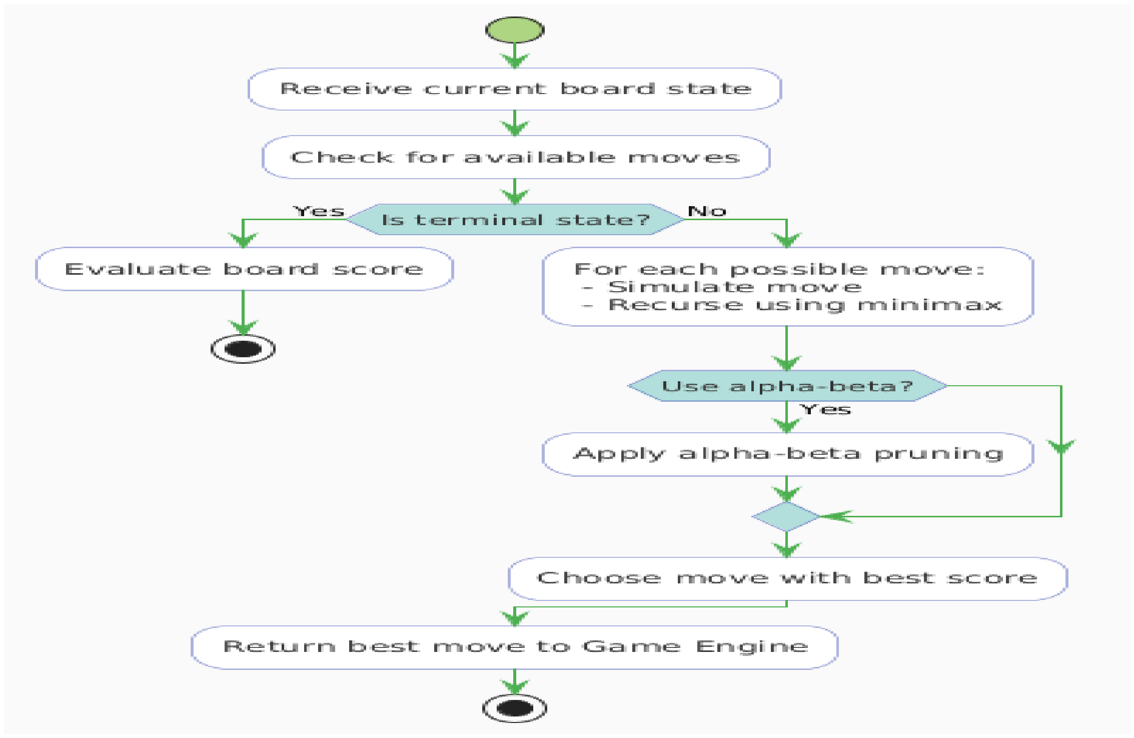
5.3 Use Case Design

A use case diagram shows the available system functionality to the user, such as registration, login, playing games, and viewing history.



5.4 AI Decision Logic

The AI uses the Minimax algorithm with alpha-beta pruning. The decision-making process, from receiving the board state to returning the best move, is described using an activity diagram.



6.Game Logic & Features

This section describes how the gameplay is implemented, covering rule enforcement, turn switching, and outcome detection.

6.1 Game Initialization

- The game initializes a Board object and two Player objects (which can be human or AI).
- The starting player is chosen (by default or randomly).

6.2 Player Interaction

- Players select cells on a 3x3 grid.
- The system checks if the move is valid using `Board::isValidMove()`.
- Valid moves are recorded using `Board::setCell()`.
- After each move, the game state is updated and drawn on the GUI.

6.3 Rule Enforcement

- The game checks for:
    - ★ **Winning condition** (3 in a row, column, or diagonal)
    - ★ **Draw condition** (board full and no winner)
- If a condition is met, the game is terminated, and the outcome is announced.

6.4 Turn Management

- The Game class maintains a variable currentTurn to alternate between players.
- After each valid move, the turn is switched using Game::switchTurn().

6.5 Modes of Play

- **Player vs Player:** Both players interact manually.
- **Player vs AI:** The human player makes a move, followed by an AI response generated by the Minimax algorithm.

6.6 GUI Integration

- The graphical interface (implemented using Qt or similar) updates the board visually after each move.
- Click events are bound to the logic functions via signal-slot connections.
- End-of-game states (win/draw) are displayed using dialog prompts.

7.AI Design

This section explains the implementation of the AI opponent in the game. The AI is designed to provide a challenging experience by making optimal decisions using game-tree search algorithms.

7.1 AI Architecture

The AI functionality is organized using a class hierarchy:

- AIAgent (defined in AIAgent.h) is an abstract base class for all AI types. It declares a virtual method makeMove(Board& board).
- MiniMaxAgent (defined in MiniMaxAgent.h/cpp) inherits from AIAgent and implements the full Minimax algorithm with alpha-beta pruning.
- NoAgent (defined in NoAgent.h) is a dummy AI agent used when AI is not required (e.g., for Player vs Player mode).

This modular approach allows swapping different AI strategies easily.

7.2 Minimax Algorithm with Alpha-Beta Pruning

The MiniMaxAgent class evaluates all possible moves using the Minimax algorithm. Key steps:

1. **Terminal State Detection**: Check if the current board state is a win, loss, or draw.
2. **Recursive Search**: Explore future board states by simulating valid moves.
3. **Scoring**: Assign scores to terminal states (e.g., +10 for win, -10 for loss, 0 for draw).
4. **Alpha-Beta Pruning**: Avoid unnecessary branches to reduce computation time.

7.3 AI Integration

- When playing Player vs AI mode, the human makes a move first.
- Then MiniMaxAgent::makeMove() is invoked to choose and execute the AI's best move.
- The board is updated and the game proceeds with the AI's decision reflected.

7.4 Customization and Extensibility

- Developers can add new agents by inheriting from AIAgent and implementing makeMove().
- This makes the AI module extensible for future upgrades (e.g., Monte Carlo Tree Search, reinforcement learning, etc.).

8.User Authentication

This section details the system's secure user authentication and management features, ensuring each player has a personalized and protected experience.

8.1 Registration and Login

- The system allows new users to register with a unique username and password.
- During login, credentials are verified securely before granting access.
- All authentication logic is managed through the UserManager class.

8.2 Password Security

- Passwords are hashed using a secure cryptographic hash function before being stored.
- The system compares hashes during login, avoiding plain-text storage entirely.
- This prevents password leakage even if the storage system is compromised.

8.3 Session Management

- Once logged in, users remain authenticated for the duration of their session.
- The system maintains the active user's identity during gameplay and history storage.

8.4 Integration with Game History

- Each user's game results are tied to their profile.
- Users can only view their own history, reinforcing personal data isolation.

8.5 Input Validation and Security

- The system validates user inputs for length, allowed characters, and duplicates.
- Defensive coding practices are used to avoid injection attacks and other exploits.

8.6 Extensibility

- The authentication module is structured for future extensions like:
  - ★ Email-based recovery
  - ★ Multi-factor authentication (MFA)
  - ★ User avatars and profiles

9.Game History

This section outlines how the game system records and manages the history of played games for each user, enabling them to review and replay past sessions.

9.1 History Storage Design

- The GameHistory class manages the saving and retrieval of completed games.
- Game data includes:
  - ★ Players' usernames
  - ★ Final board state

    - ★ Result (win/loss/draw)
    - ★ Timestamp
- The data is stored using either:
    - ★ A lightweight database like **SQLite**, or
    - ★ A structured file (e.g., JSON, custom format)

9.2 Saving Game Sessions

- After each completed game, the GameHistory::saveGame() function is called.
- This ensures that the result is persistently stored and associated with the current user.

9.3 Viewing Game History

- Users can view a list of all previously played games.
- Each entry displays the opponent, outcome, and date played.
- Clicking a history item allows users to replay the game board step-by-step.

9.4 Replay Feature

- Replay functionality is implemented by iterating over stored move sequences.
- The GUI reconstructs the board state visually to simulate how the game unfolded.

10.Testing Strategy

This section outlines the testing practices applied to ensure the reliability, correctness, and robustness of the Tic Tac Toe game.

10.1 Purpose

To validate the behavior of the system through both automated testing using Google Test and manual gameplay testing.

10.2 Test Scope

We tested the following modules thoroughly:

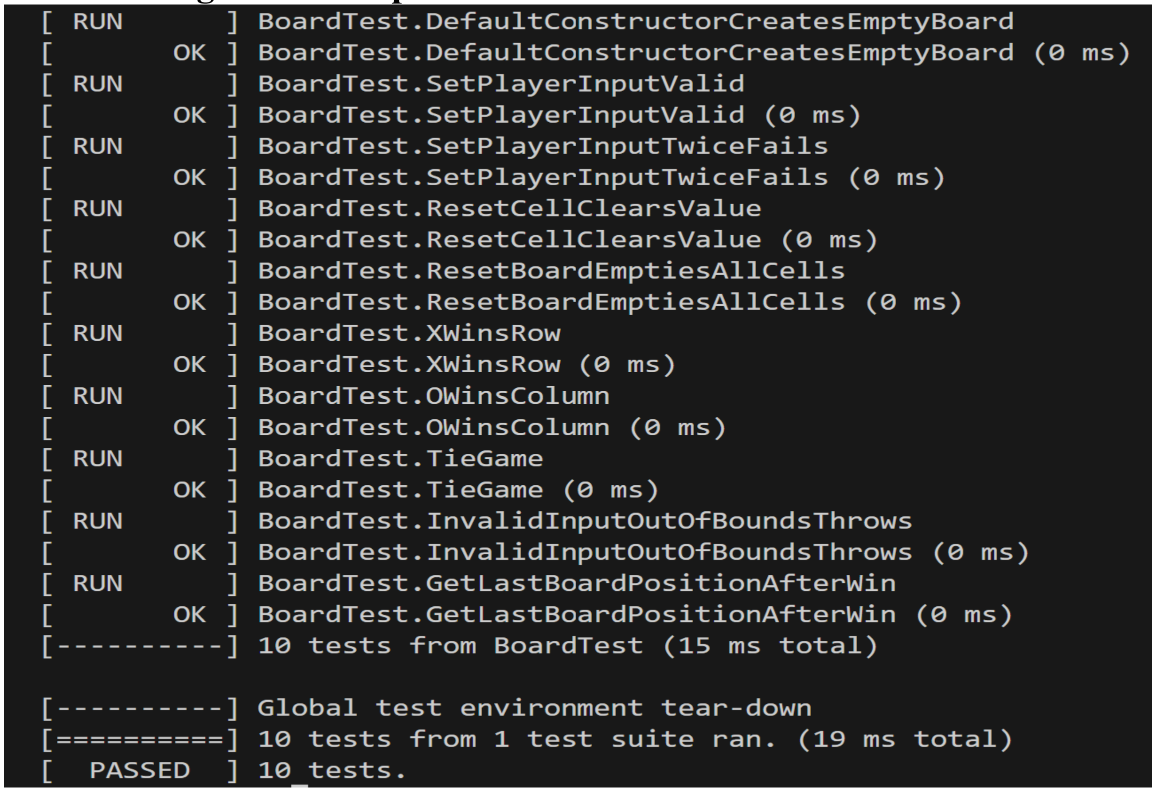
- **Game Logic (Board)**: win detection, valid moves, board reset
- **AI Agent (MiniMaxAgent)**: move selection, blocking opponent, scoring
- **User History (PlayerHistory)**: file-based game recording and statistics
- **Game Integration**: full game flow and GUI behavior (manually verified)

10.3 Tools and Frameworks

- **Google Test**: For automated unit testing in C++
- **Visual Studio Code + CMake + MinGW**: For test development
- **Manual GUI testing**: For user interface functionality

10.4 Test Examples & Screenshots

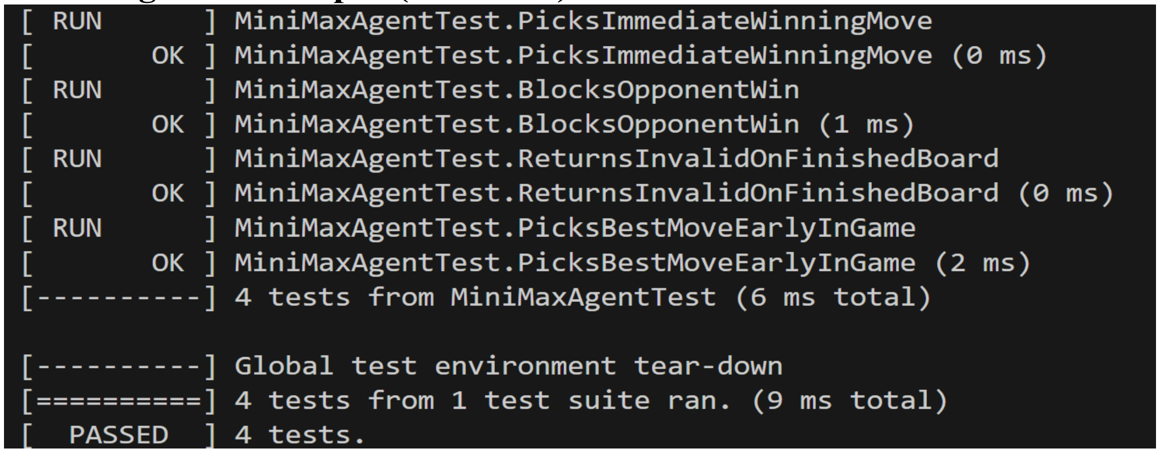
✔ **Board Logic Test Output**



```
[ RUN      ] BoardTest.DefaultConstructorCreatesEmptyBoard
[       OK ] BoardTest.DefaultConstructorCreatesEmptyBoard (0 ms)
[ RUN      ] BoardTest.SetPlayerInputValid
[       OK ] BoardTest.SetPlayerInputValid (0 ms)
[ RUN      ] BoardTest.SetPlayerInputTwiceFails
[       OK ] BoardTest.SetPlayerInputTwiceFails (0 ms)
[ RUN      ] BoardTest.ResetCellClearsValue
[       OK ] BoardTest.ResetCellClearsValue (0 ms)
[ RUN      ] BoardTest.ResetBoardEmptiesAllCells
[       OK ] BoardTest.ResetBoardEmptiesAllCells (0 ms)
[ RUN      ] BoardTest.XWinsRow
[       OK ] BoardTest.XWinsRow (0 ms)
[ RUN      ] BoardTest.OWinsColumn
[       OK ] BoardTest.OWinsColumn (0 ms)
[ RUN      ] BoardTest.TieGame
[       OK ] BoardTest.TieGame (0 ms)
[ RUN      ] BoardTest.InvalidInputOutOfBoundsThrows
[       OK ] BoardTest.InvalidInputOutOfBoundsThrows (0 ms)
[ RUN      ] BoardTest.GetLastBoardPositionAfterWin
[       OK ] BoardTest.GetLastBoardPositionAfterWin (0 ms)
[----------] 10 tests from BoardTest (15 ms total)

[----------] Global test environment tear-down
[==========] 10 tests from 1 test suite ran. (19 ms total)
[  PASSED  ] 10 tests.
```

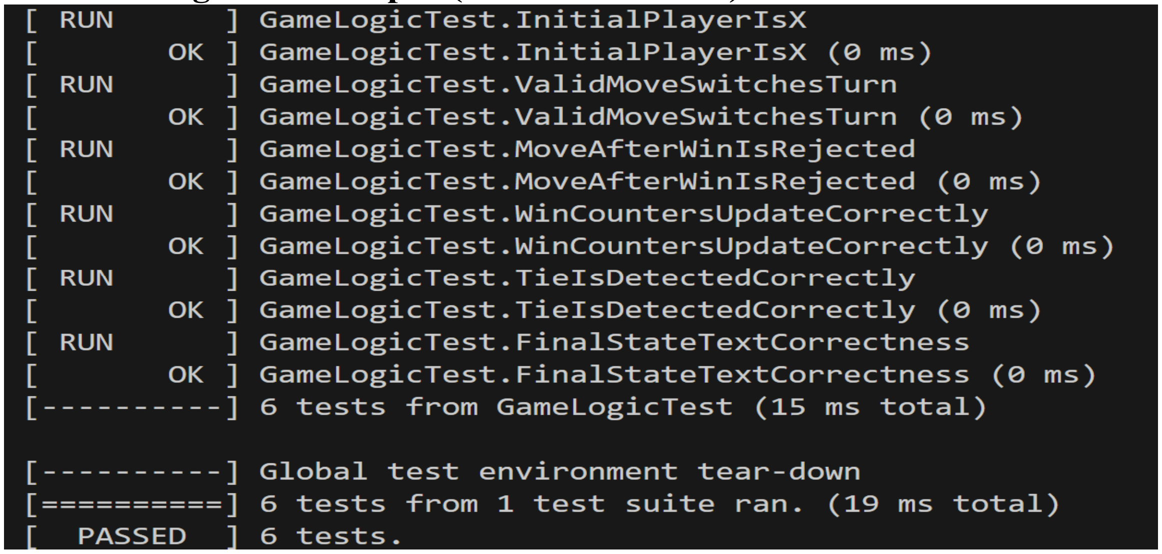
✓ AI Logic Test Output (MiniMax)



```
[ RUN      ] MiniMaxAgentTest.PicksImmediateWinningMove
[       OK ] MiniMaxAgentTest.PicksImmediateWinningMove (0 ms)
[ RUN      ] MiniMaxAgentTest.BlocksOpponentWin
[       OK ] MiniMaxAgentTest.BlocksOpponentWin (1 ms)
[ RUN      ] MiniMaxAgentTest.ReturnsInvalidOnFinishedBoard
[       OK ] MiniMaxAgentTest.ReturnsInvalidOnFinishedBoard (0 ms)
[ RUN      ] MiniMaxAgentTest.PicksBestMoveEarlyInGame
[       OK ] MiniMaxAgentTest.PicksBestMoveEarlyInGame (2 ms)
[----------] 4 tests from MiniMaxAgentTest (6 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (9 ms total)
[  PASSED  ] 4 tests.
```

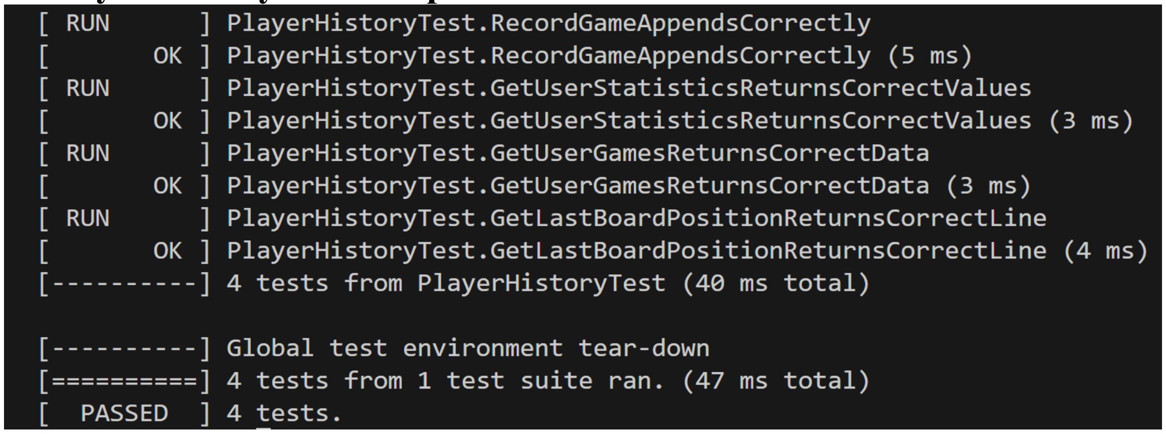
✓ Game Logic Test Output (TicTacToeGame)



```
[ RUN      ] GameLogicTest.InitialPlayerIsX
[       OK ] GameLogicTest.InitialPlayerIsX (0 ms)
[ RUN      ] GameLogicTest.ValidMoveSwitchesTurn
[       OK ] GameLogicTest.ValidMoveSwitchesTurn (0 ms)
[ RUN      ] GameLogicTest.MoveAfterWinIsRejected
[       OK ] GameLogicTest.MoveAfterWinIsRejected (0 ms)
[ RUN      ] GameLogicTest.WinCountersUpdateCorrectly
[       OK ] GameLogicTest.WinCountersUpdateCorrectly (0 ms)
[ RUN      ] GameLogicTest.TieIsDetectedCorrectly
[       OK ] GameLogicTest.TieIsDetectedCorrectly (0 ms)
[ RUN      ] GameLogicTest.FinalStateTextCorrectness
[       OK ] GameLogicTest.FinalStateTextCorrectness (0 ms)
[----------] 6 tests from GameLogicTest (15 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (19 ms total)
[  PASSED  ] 6 tests.
```

✓ Player History Test Output



```
[ RUN      ] PlayerHistoryTest.RecordGameAppendsCorrectly
[       OK ] PlayerHistoryTest.RecordGameAppendsCorrectly (5 ms)
[ RUN      ] PlayerHistoryTest.GetUserStatisticsReturnsCorrectValues
[       OK ] PlayerHistoryTest.GetUserStatisticsReturnsCorrectValues (3 ms)
[ RUN      ] PlayerHistoryTest.GetUserGamesReturnsCorrectData
[       OK ] PlayerHistoryTest.GetUserGamesReturnsCorrectData (3 ms)
[ RUN      ] PlayerHistoryTest.GetLastBoardPositionReturnsCorrectLine
[       OK ] PlayerHistoryTest.GetLastBoardPositionReturnsCorrectLine (4 ms)
[----------] 4 tests from PlayerHistoryTest (40 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (47 ms total)
[  PASSED  ] 4 tests.
```

10.5 Test Summary

- **Unit Tests Passed**: 100%
- **Code Behavior Verified**: ✓
- **Manual GUI Walkthrough**: Validated win/loss/tie recognition, AI interaction, login flow.
- **Test Coverage**: Focused on all major modules with multiple logic branches tested.

10.6 Future Testing Considerations

- Use QtTest for automated GUI testing
- Add mock storage layer for headless history testing
- Expand to simulate full multiplayer testing sessions

11.CI/CD Pipeline

Continuous Integration and Continuous Deployment (CI/CD) was implemented using GitHub and GitHub Actions to automate the testing and deployment process for the project.

11.1 Tools Used

- **GitHub**: Version control and remote collaboration
- **GitHub Actions**: CI pipeline for automated testing
- **CMake + Google Test**: Integration with GitHub Actions for test runs

11.2 CI/CD Workflow

1. **Code Push or Pull Request:** Any code change pushed to the repository triggers the CI pipeline.
2. **Build Stage:** The CMake system configures and compiles the project.

3. **Test Stage:** Google Test is executed to run unit and integration tests.
4. **Result Stage:** Build and test results are shown directly in the GitHub PR interface.

11.3 Benefits

- Automated test runs prevent integration of broken code
- Early detection of errors before merging
- Simplified testing across machines (Linux and Windows environments)

11.4 Example Workflow Snippet (GitHub Actions)

```
name: C++ CI
on: push: branches: [ main ] pull_request: branches: [ main ]
jobs: build: runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v2
- name: Install Dependencies
  run: sudo apt-get install cmake g++
- name: Configure
  run: cmake -S . -B build
- name: Build
  run: cmake --build build
- name: Run Tests
  run: ./build/playerhistory_tests && ./build/board_tests && ./build/minimax_tests
```

11.5 Future Improvements

- Add test coverage badges
- Trigger deployments to cloud platforms (if GUI or API hosted)
- Enforce code linting using clang-format in pipeline

12.Performance and Optimization

This section outlines how performance was monitored and optimized during the development of the game.

12.1 Performance Metrics

- **Response Time**: Measured how quickly the system responds to user input or AI moves (target: < 200 ms).
- **Memory Usage**: Evaluated memory consumption of the game state and AI during gameplay.
- **CPU Utilization**: Monitored especially during AI decision-making to ensure efficiency.

12.2 Optimization Techniques

- **Alpha-Beta Pruning**: Applied to the Minimax algorithm to significantly reduce the number of nodes evaluated.
- **Efficient Data Structures**: Use of arrays, hash tables, and minimal dynamic allocation helped reduce overhead.
- **Code Profiling**: Tools such as gprof or valgrind (if used) helped identify slow or memory-intensive code paths.

12.3 Results

- The AI's average move time stayed well under 150 ms even in complex board states.
- The application maintained low memory and CPU usage under typical conditions.
- Optimized logic ensured quick board updates and minimal latency in the GUI.

12.4 Potential Improvements

- Introducing multithreading for AI computation in future versions.
- Saving precomputed board evaluations to further speed up decision-making.
- Adopting hardware acceleration (e.g., GPU) for rendering if performance becomes a bottleneck.

13.References

[1] G. Inc., “GoogleTest: Google Testing and Mocking Framework,” [Online]. Available: https://github.com/google/googletest. [Accessed: Apr. 10, 2025].

[2] PlantUML, “PlantUML - Visualizing UML diagrams,” [Online]. Available: https://plantuml.com. [Accessed: Apr. 10, 2025].

[3] GitHub Docs, “GitHub Actions Documentation,” [Online]. Available: https://docs.github.com/en/actions. [Accessed: Apr. 10, 2025].

[4] Qt Company, “Qt for Application Development,” [Online]. Available: https://www.qt.io. [Accessed: Apr. 10, 2025].

[5] cppreference.com, “C++ Standard Library Reference,” [Online]. Available: https://en.cppreference.com. [Accessed: Apr. 10, 2025].

[6] ChatGPT, OpenAI, “Code assistance and technical writing,” Used with instructor's permission during project development, Apr. 2025.

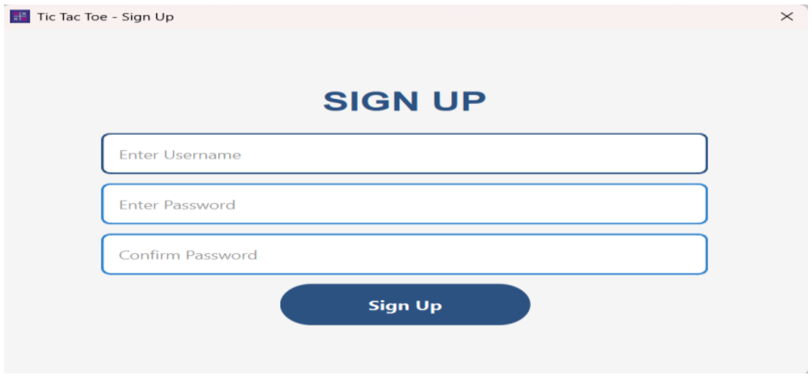
[7] Valgrind Project, “Valgrind: Debugging and Profiling Tools,” [Online]. Available: https://valgrind.org. [Accessed: Apr. 10, 2025].

[8] The LLVM Compiler Infrastructure, “Clang Static Analyzer,” [Online].Available: https://clang-analyzer.llvm.org. [Accessed: Apr. 10, 2025].

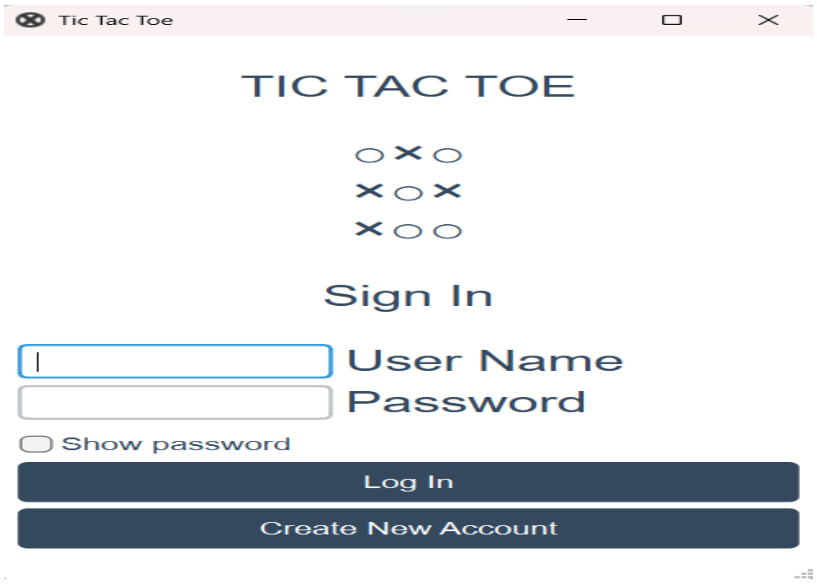
14.Appendices

The following images provide a complete walkthrough of the user experience in the Tic Tac Toe game, beginning from sign-up to gameplay and viewing history:

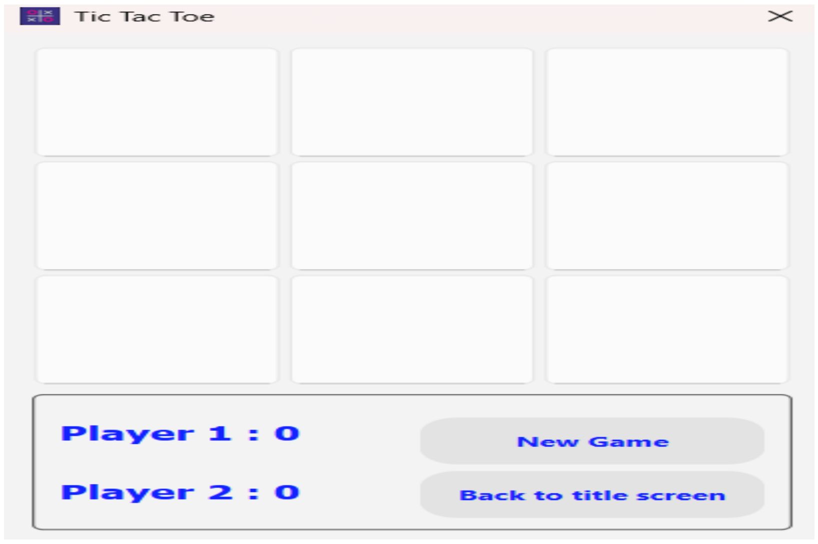
- ★ **Figure 1: Sign-up Screen**



- ★ **Figure 2: Login Interface**



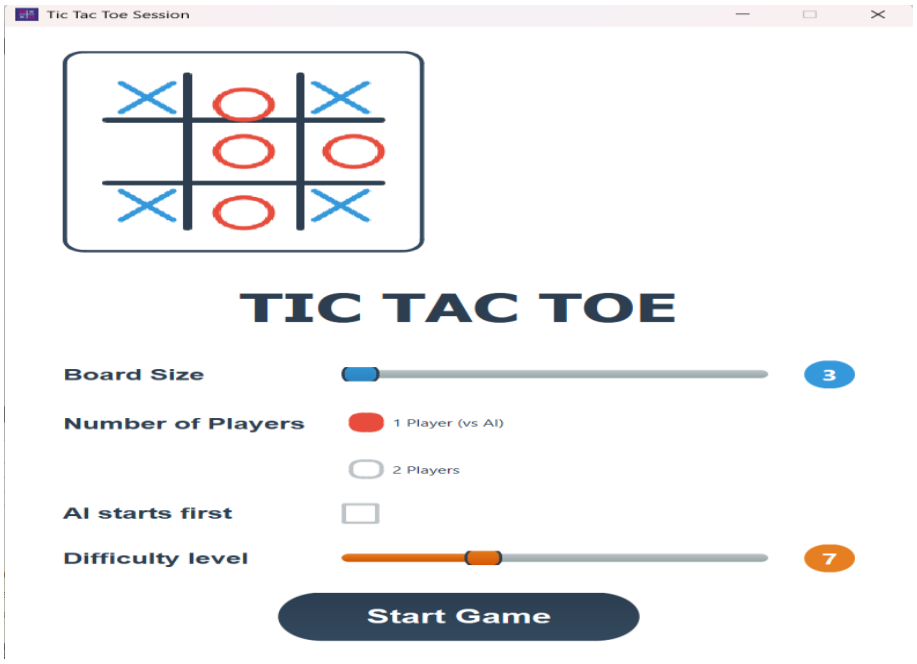
★ **Figure 3: Game Board View**



Player 1 : 0

Player 2 : 0

★ **Figure 4: AI Gameplay Mode**



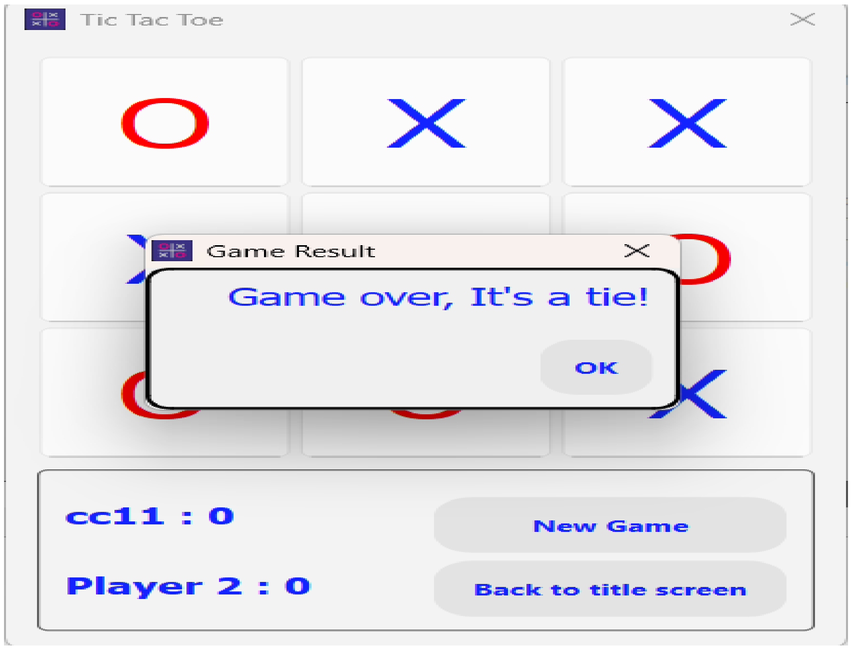
Board Size: 3

Number of Players: [✓ 1 Player (vs AI)] 2 Players

AI starts first: ☐

Difficulty level: 7

★ **Figure 5: Game Result Popup**

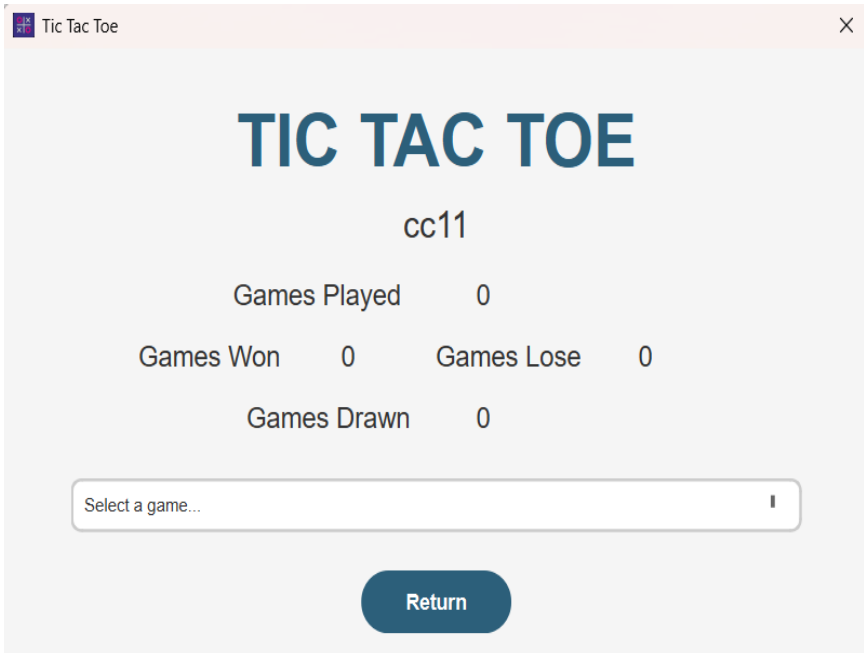


Game over, It's a tie!

cc11 : 0

Player 2 : 0

★ **Figure 6: History Page with Game Stats**



cc11

Games Played 0

Games Won 0 Games Lose 0

Games Drawn 0

These screenshots demonstrate the successful integration between UI components, logic layers, and history tracking. Each step highlights how users interact with the system and how the system responds in real time.