Software Design Specification

Version: 1.0

Advanced Tic-Tac-Toe Game

Submitted by

Your Next Move Team

Team members:

Name	Section	ID
Mariam Ahmed Hamed	4	9220812
Mariam Mohamed Saeed	4	9220828
Malak Waleed Fouad	4	9220867
Menna Moutaz Elhosseiny	4	9220874
Maha Yasser Mohamed	4	9220878

Submitted to

Dr. Omar Nasr

Table of contents:

1. Introduction

1.1 Purpose:

The purpose of this document is to provide a detailed description of the design and architecture of our advanced Tic-Tac-Toe game. This document will serve as a guide which shows the implementation steps and the system components connections.

1.2 Scope:

This document covers the complete design of the Tic-Tac-Toe game, including game logic, AI opponent, GUI, user authentication, personalized game history, testing strategies, performance optimization, and CI/CD integration.

1.3 Definitions, Acronyms and Abbreviations:

AI: Artificial Intelligence

GUI: Graphical User Interface

SDS: Software Design Specification

CI/CD: Continuous Integration/Continuous Deployment

Git: Version control system

SQLite: Lightweight database management system

1.4 References:

[1] "GeeksforGeeks," 16 Janruary 2023. [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/. [Accessed 25 June 2024].

[2] "Great Learning," 30 April 2024. [Online]. Available: https://www.mygreatlearning.com/blog/alpha-beta-pruning-in-ai/. [Accessed 25 June 2024].

[3] "GeeksforGeeks," 13 June 2022. [Online]. Available: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/. [Accessed 25 June 2024].

[4] "SQLite," [Online]. Available: https://www.sqlite.org/cintro.html. [Accessed 25 June 2024].

[5] KDAB, "Unit Testing from Qt Creator," 22 March 2022. [Online]. Available: https://www.youtube.com/watch?v=N4pvvCToogM. [Accessed 25 June 2024].

[6] "ChatGPT".

[7] "GitHub," [Online]. Available: https://education.github.com/git-cheat-sheet-education.pdf. [Accessed 25 June 2024].

1.5 Overview:

This SDS document is structured to provide a comprehensive view of the system's architecture and design. It starts with a high-level overview and then dives into detailed design specifications, data structures, user interfaces, testing strategies, performance optimization, and version control.

2. System Architecture:

- Game Logic Engine: manages the core mechanics of the game, including player moves, turns alternating, win checking and game state evaluation.
- AI Module: implements the AI opponent using the minimax algorithm with alpha-beta pruning to make the most optimal moves.

- GUI Module: provides the graphical user interface for users to play the game, view their profiles, and view game history.
- User Authentication Module: manages user accounts, including secure login and registration processes.
- Game History Module: stores and retrieves personalized game histories.
- Database: SQLite database for storing user data and game histories.
- CI/CD Pipeline: GitHub Actions for automated testing and deployment.

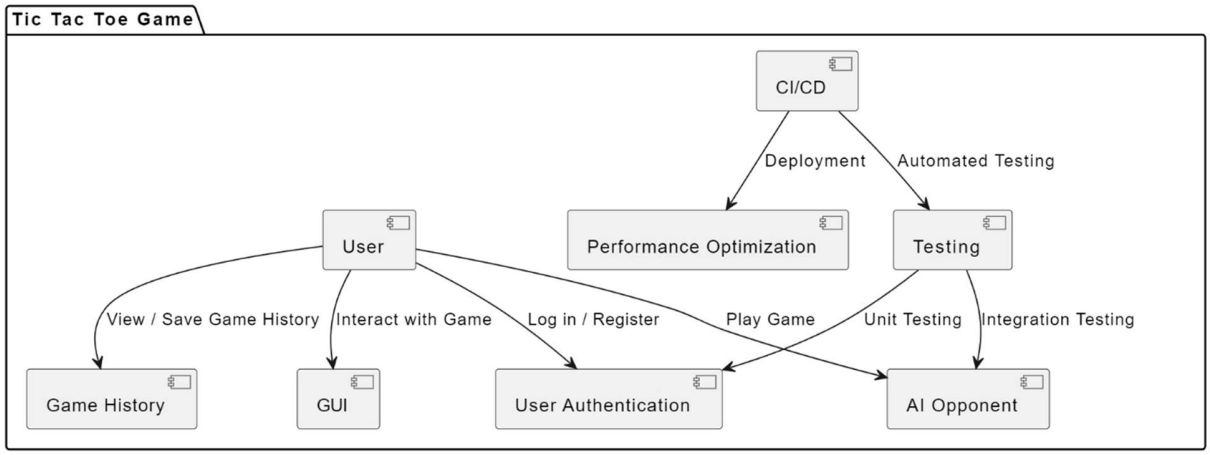


Figure 1 High-Level System Architecture

3. Design Considerations

3.1 Assumptions and Dependencies:

- Players should have access to a computer with the necessary operating system
- Developers should have access to the required development tools and resources.

3.2 System Environment:

- Operating Systems: Windows
- Development Tools: C++ compiler, Qt framework, SQLite
- Version Control: Git with GitHub
- CI/CD Tools: GitHub Actions

4. Detailed System Design

4.1 Functions:

4.1.1 User Authentication:

- `signup(sqlite3* db, const std::string& email, const std::string& password, const std::string& name, int age, const std::string& city)`

Handles the signup process, adding a new user to the database.

- `login(sqlite3* db, const std::string& email, const std::string& password)`

Handles the login process, authenticating a user based on the provided credentials.

- `onLoginButtonClicked()`

Handles the login button click event.

- `onSignupButtonClicked()`

Handles the signup button click event.

- `onPlayer2LoginButtonClicked()`

Handles Player 2 login button click event.

- `onPlayer2SignupButtonClicked()`

Handles Player 2 signup button click event.

4.1.2 Game Management:

- `initializeGame()`

Initializes the game settings and state.

- `resetGame()`

Resets the game to its initial state.

- `resetGame1(), resetGame2()`

Function to set empty board.

- `onpgClicked()`

Handles the pg button click event.

4.1.3 Game Logic

- `checkGameState()`

Checks the current state of the game to determine if there is a win or a draw.

- `askPlayAgain(const QString& result)`

Asks the players if they want to play again after a game ends.

4.1.4 AI Opponent:

- `makeAIMove()`

Makes a move for the AI player using the AIPlayer class.

4.1.5 Database Operations:

- `verifyTablesExist()`

Verifies if the necessary tables exist in the database.

- `createTablesIfNeeded()`

Creates tables in the database if they do not exist.

- `loadUserData(const std::string &email)`

Loads user data from the database based on the provided email.

- `saveMove(int gameId, const GameBoard& board, const std::string& playerTurn, int moveNumber, sqlite3* db)`

Saves a move to the database.

- `getPlayerGameIds(const std::string& playerEmail, sqlite3* db)`

Retrieves the game IDs associated with a player.

- `getGameDetails(int gameId, sqlite3* db)`

Retrieves the game details for a specific game ID.

4.1.6 User Interface (UI) Updates:

- `showBoard(const char boardStr[]),showBoard1(const char boardStr[])`

Displays the current state of the game board.

- `updateBoardUI()`

Updates the UI to reflect the current state of the board.

- `updateTurnLabel()`

Updates the UI label to show the current player's turn.

- `showMoveByMove(int gameId, const std::string& playerEmail)`

Shows the game move by move for a specific game ID.

- `showNextMove()`

Shows the next move in the replay.

- `showPlayerGameIds()`

Displays the game IDs associated with the current player.

- `showGameOptions(const std::string& playerEmail, int gameId)`

Displays game options for a specific game ID.

- `showFinalMove(int gameId)`

Shows the final move of a game.

- `showGameDetails`(int gameId)

Shows the details of a specific game.

- `n_showPasswordCheckBox_stateChanged(int state)`

Handles the state change event of the "show password" checkbox for the main player.

- `on_showPassword1CheckBox_stateChanged(int state)`

Handles the state change event of the "show password" checkbox for Player 2.

- `showPlayer1Stats()`

Displays the statistics for Player 1.

- showPlayer2Stats()

Displays the statistics for Player 2.

4.1.7 Player Statistics:

- getPlayerStats(const std::string& email,int& pvp_win_count, int& pvp_lose_count, int& pvp_total_games, int& pve_win_count, int& pve_lose_count, int& pve_total_games, int& total_wins, int& total_loss, int& total_games)

Retrieves the statistics for a specific player.

- updatePlayerStats(const std::string& email, int pvp_win_count, int pvp_lose_count, int pvp_total_games, int pve_win_count, int pve_lose_count, int pve_total_games)

Updates the statistics for a specific player.

- handleGameOutcome(const std::string& player1Email, const std::string& player2Email, int res)

Handles the outcome of the game based on the result.

4.1.8 Utility Functions:

- handleButtonClick()

Handles a button click on the game board.

- getLoggedInPlayerEmail()

Retrieves the email of the currently logged-in player.

- getPlayer2Email()

Retrieves the email of Player 2.

- timeToString(std::chrono::system_clock::time_point timePoint)

Converts a time point to a string representation.

- `customHash(const std::string& str)`

Generates a custom hash for the given string.

- `hashPassword(const std::string& password)`

Hashes the given password for secure storage.

4.1.9 Frame Switches:

- `onreturn1Clicked()`

Handles the return button click (1).

- `onreturn2Clicked()`

Handles the return button click (2).

- `onreturn_2Clicked()`

Handles the return button click (_2).

- `onreturn3Clicked()`

Handles the return button click (3).

- `onreturn4Clicked()`

Handles the return button click (4).

- `onreturn5Clicked()`

Handles the return button click (5).

- `onreturn6Clicked()`

Handles the return button click (6).

- `onSwitchToSignupButtonClicked()`

Switches the UI to the signup frame.

- onSwitchToLoginButtonClicked()

Switches the UI to the login frame.

- onSwitchToPlayer2SignupButtonClicked()

Switches the UI to the Player 2 signup frame.

- onSwitchToPlayer2LoginButtonClicked()

Switches the UI to the Player 2 login frame

4.1.10 Game Modes:

- onPLAYClicked()

Handles the PLAY button click event to start the game.

- onPVPButtonClicked()

Handles the Player vs Player button click event.

- onPVEButtonClicked()

Handles the Player vs Environment (AI) button click event.

4.1.11 Log out:

- onlogoutClicked()

Handles the logout button click event.

4.2 Classes:

- GameBoard Class

Manages the state of the game board

Member Functions:

- `int checkWin() const`

Checks the current state of the game board and determine if there is a winning condition.

- `int getValue(int row, int col) const`

Retrieves the value of a specific cell on the game board.

- `void setValue(int row, int col, int value)`

Sets the value of a specific cell on the game board.

- AIPlayer Class

Responsible for the AI's behavior and decision-making during the game.

Member Variables:

- ```
  struct TreeNode {
  GameBoard board;
   std::vector<TreeNode*> children;
   int moveRow;
  int moveCol;
   int score;
  TreeNode() : moveRow(-1), moveCol(-1), score(0) {} };
  ```

Used to represent a node in the game tree. Each node contains a game board state, possible children nodes (representing possible future states), the move that led to this state, and the score of the state.

Member Functions:

- `void makeMove(GameBoard& board) const`
  ```

Makes the best possible move on the game board based on the AI's analysis.

- `void build_tree(TreeNode* node, int player) const`

Constructs the game tree representing all possible moves from the current state.

- `int minimax(TreeNode* node, int alpha, int beta, bool is_max, int depth) const`

Applies the minimax algorithm with alpha-beta pruning to evaluate the game tree and find the best move.

- `int evaluate(const GameBoard& board) const`

Evaluates the current state of the game board and assign a score.

5 Data Design

5.1 Database Overview:

The database for the Tic-Tac-Toe game application is designed to store and manage user data, game sessions, and moves made during games. It is implemented using SQLite, a lightweight, disk-based database management system. The database is comprised of three primary tables: players, games, and moves. Each table serves a specific purpose and helps maintain the integrity and performance of the application.

5.2 Database Schema:

- **Players Table**

Stores information about registered players.

Column Name	Data Type	Constraints	Description
ID	INTEGER	PRIMARY KEY	Unique identifier for each player
Email	TEXT	UNIQUE	Email address of the player
Password	TEXT		Hashed password of the player
Name	TEXT		Name of the player
City	TEXT		City of residence of the player
Age	INTEGER		Age of the player
PVP_win_count	INTEGER	DEFAULT 0	Number of player-vs-player wins
PVP_lose_count	INTEGER	DEFAULT 0	Number of player-vs-player losses
PVP_total_games	INTEGER	DEFAULT 0	Number of player-vs-player games
PVE_win_count	INTEGER	DEFAULT 0	Number of player-vs-AI wins
PVE_lose_count	INTEGER	DEFAULT 0	Number of player-vs-AI losses
PVE_total_games	INTEGER	DEFAULT 0	Number of player-vs-AI games
Total_games	INTEGER	DEFAULT 0	Total number of games played
Current_date	TEXT		Date of player's current activity
Last_login_date	TEXT		Date of player's last login

- **Games Table**

Stores information about game sessions.

Column Name	Data Type	Constraints	Description
ID	INTEGER	PRIMARY KEY AUTOINCREMENT	Unique identifier for each game
Player1_email	TEXT	NOT NULL	Email address of Player 1
Player2_email	TEXT	NOT NULL	Email address of Player 2
Date	TEXT	NOT NULL	Date and time of the game
Game_mode	INTEGER	NOT NULL	Game mode identifier (player-vs-player or player-AI)

- **Moves Table**

Stores the moves made during each game session.

Column Name	Data Type	Constraints	Description
ID	INTEGER	PRIMARY KEY AUTOINCREMENT	Unique identifier for each move
Game_id	INTEGER	NOT NULL	Foreign key referencing games(id)
Board	TEXT	NOT NULL	State of the game board
Player_turn	TEXT	NOT NULL	Email address of the player whose turn it is
Move_number	INTEGER	NOT NULL	Sequence number of the move

ER diagram representing the database design

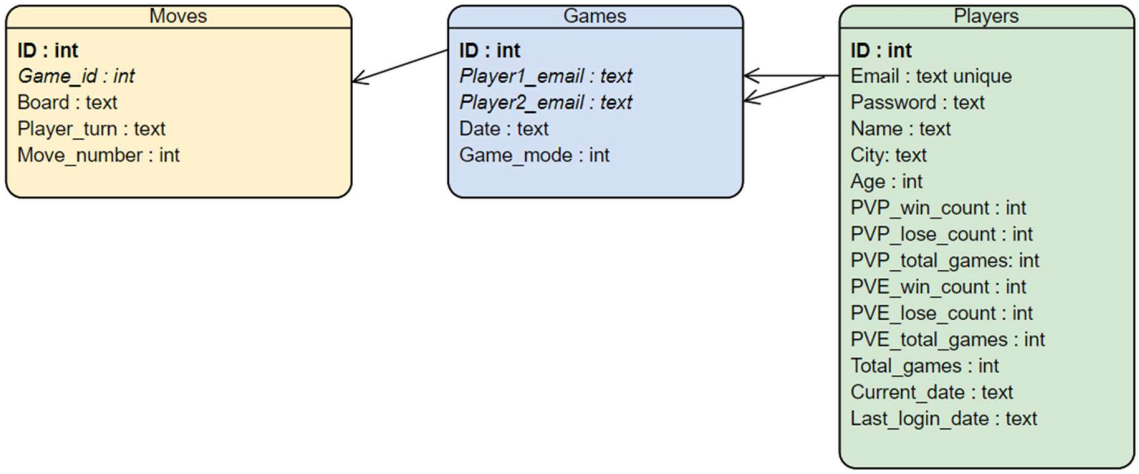


Figure 2 ER Diagram

6 Human Interface Design

6.1 User Interface Overview:

The Tic-Tac-Toe application features a straightforward and user-friendly interface designed to provide a seamless gaming experience. It begins with a Welcome Screen, where users can choose to either log in or sign up. The Login Screen allows existing users to enter their email and password, while the Sign Up Screen enables new users to create an account by entering their email, password, name, and age. After a successful login, users are taken to the main menu, which offers options for "Player vs Player", "Player vs AI", and "View Stats". The Game Board interface is where the actual Tic-Tac-Toe game is played, with separate boards for "Player vs Player" and "Player vs AI" modes. Additionally, the Stats Screen displays player statistics such as total games played, wins, and losses. The UI is designed to guide users smoothly from one screen to another, ensuring an intuitive and enjoyable experience.

6.2 Screen Definitions:

- Frame 1: welcome frame and registration options
- Frame 2: optional frame if the user chooses to sign up
- Frame 3: log in operation
- Frame 4: display player 1's information (play button goes to frame 5 and game ids button goes to frame 7)
- Frame 5: game mode options
- Frame 6.1 : registration options for player 2 if the user chooses player-vs-player mode
- Frame 6.1.1: optional frame if the user chooses to sign up for player 2
- Frame 6.1.2: log in operation for player 2
- Frame 6.1.3: player-vs-player game board
- Frame 6.2: player-vs-AI game board
- Frame 7: display game history

7 Testing Strategy

7.1 User Interface Overview:

For Unit Testing, we used QTest due to its high its accuracy and efficiency. We applied many test cases for our functions including functions that are responsible for accurately implementing the game logic, checking for the winner and updating the game board with the user's moves. We also applied tests to functions that communicate with our database in order to test it such as the functions responsible for the login and signup of the players. All the tests implemented passed with no errors verifying the implementation of our program.

7.2 Function Tests:

- `testAIMove()`

  Validates the accuracy of the outputs of the makeMove() function through 2 different test cases

- `testBuildTree()`

  Validates the accuracy of the outputs of the build_tree() function

- `testMinimax()`

  Validates the accuracy of the outputs of the minimax() function through 5 different test cases

- `testEvaluate()`

  Validates the accuracy of the outputs of the evaluate() function through 7 different test cases to test the winning of either AI or the player and the draw between them

- `testCheckWin()`

Validates the accuracy of the outputs of the checkWin() function through 9 different test cases to test the winning of either player 1 or 2 and the case of a draw

- `void testGameNotOver()`

Tests the behavior of the game when the game is not over yet

- `void testInitialEmptyBoard()`

Tests the behavior of the game when the board is still empty

- `void testSetGetValue()`

Validates the accuracy of the outputs of the setValue() and getValue() functions

- `void testOutOfBoundsAccess()`

Tests the behavior of the program when an out of bound value is entered on the board (e.g.:3)

- `void testLogin()`

Tests the behavior of the login() function

- `void testSignup()`

Tests the behavior of the signup() function

- `void testCustomHash(), void testHashPassword()`

Tests the effiency and accuracy of the hashing functions hashPassword() and customHash()

8 Performance Optimization

8.1 Metrics to be Monitored:

- Response Time: Time taken by AI to make a move.
- Memory Usage: Memory consumption during gameplay.
- CPU Utilization: CPU usage during database calculations.

8.2 Optimization Strategies:

- Optimize the minimax algorithm using alpha-beta pruning.
- Efficient memory management in game state storage and retrieval.

9 Version Control and CI/CD

9.1 Version Control Strategy:

- Use Git for version control.
- Branching strategy: main for stable releases and other branches, one for each developer.

9.2 CI/CD Pipeline Configuration:

- GitHub Actions: Set up workflows for building and deploying the application.
- Build Workflow: Compile the code and run unit tests.
- Deploy Workflow: Deploy the application on successful tests.

10 Appendices

10.1 Sequence Diagram Figure:

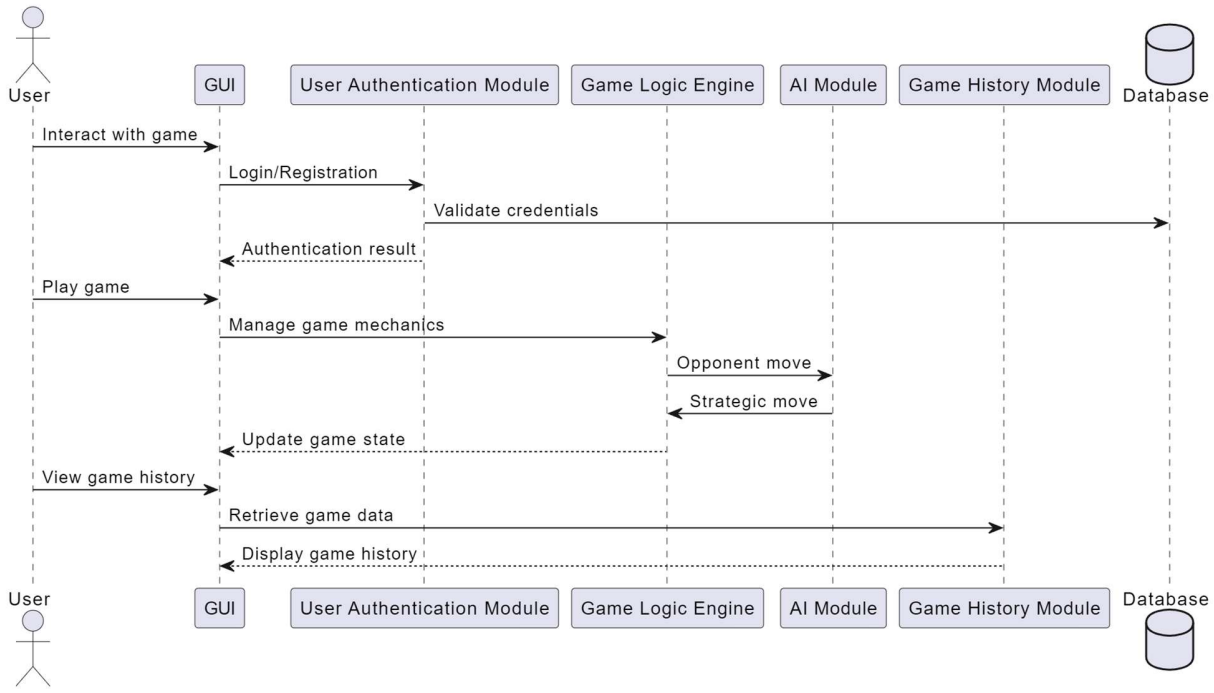


Figure 3 Sequence Diagram

10.2 Design Flow Diagram Figure:

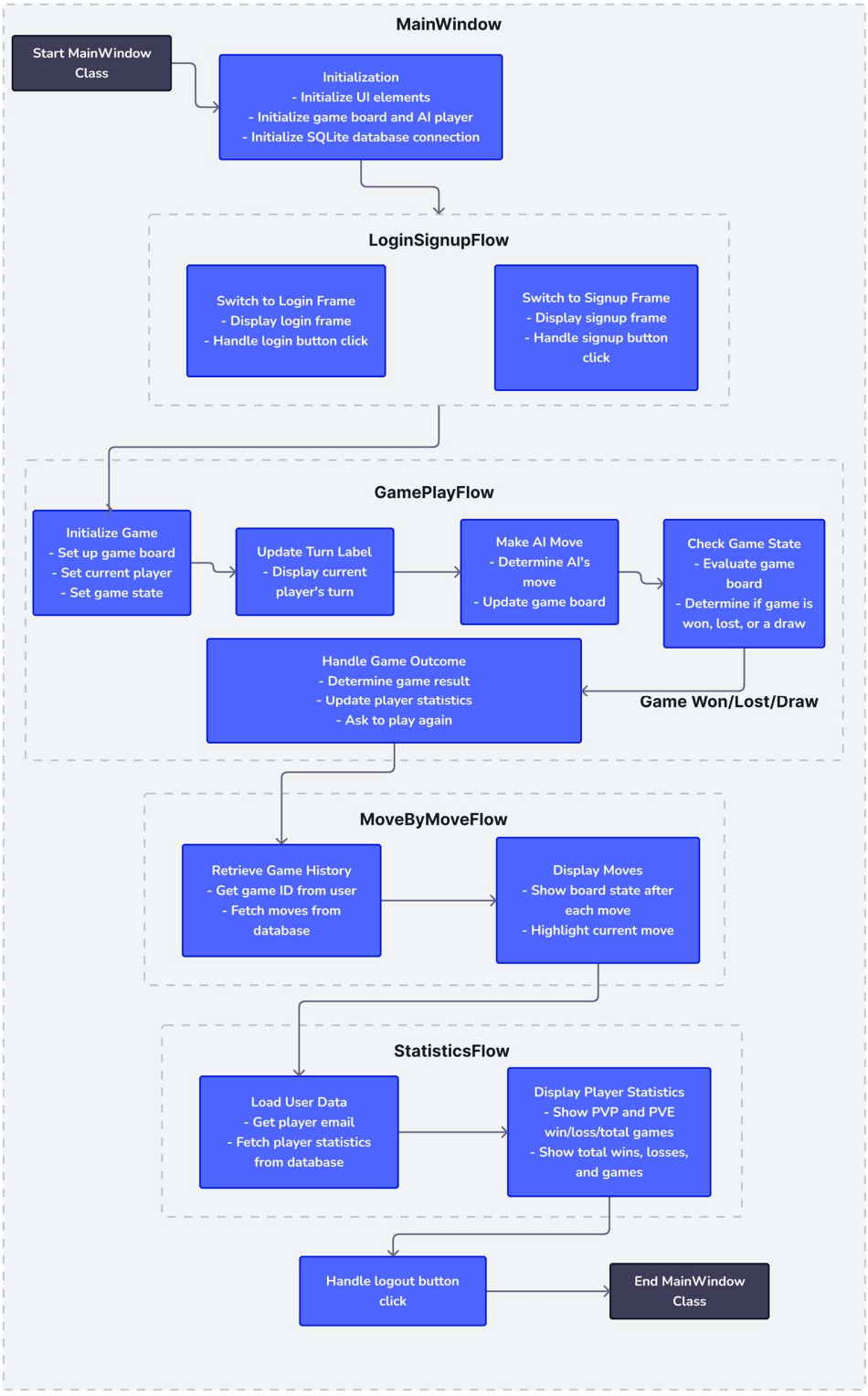
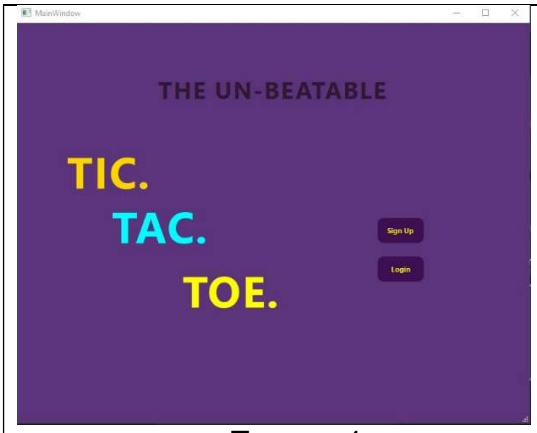
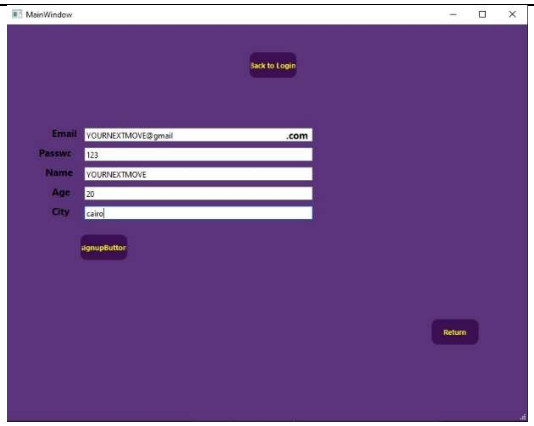


Figure 4 Design Flow Diagram

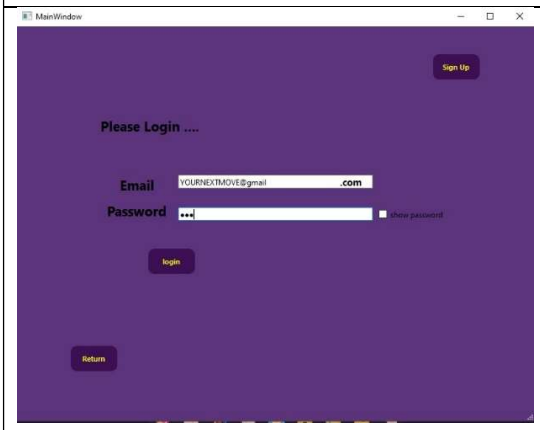
10.3 Frames:



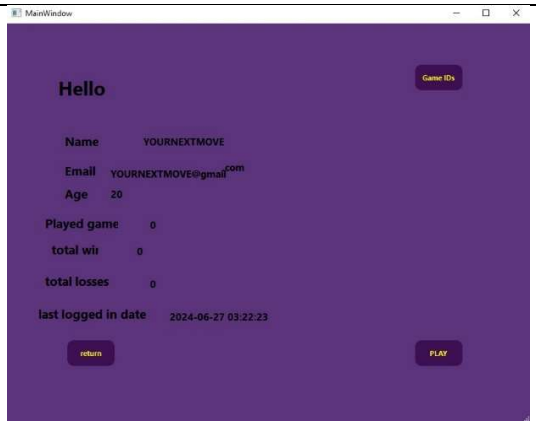
Frame 1



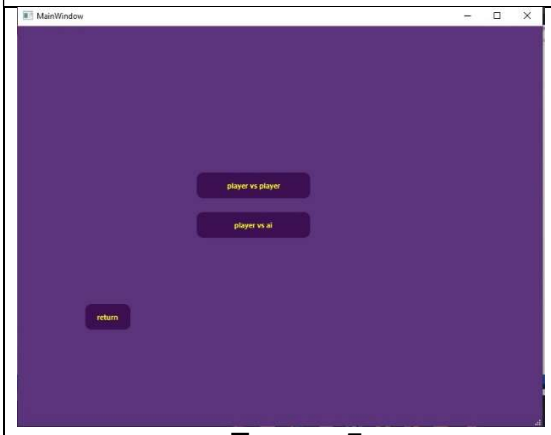
Frame 2



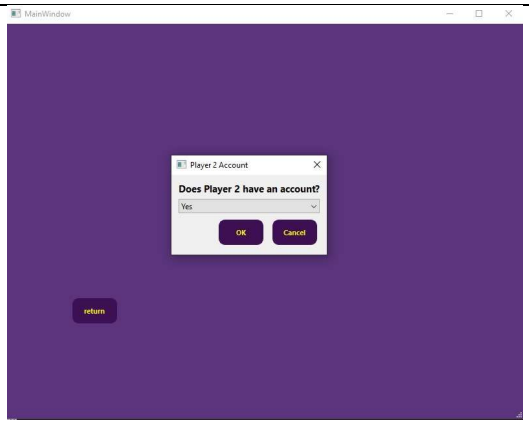
Frame 3



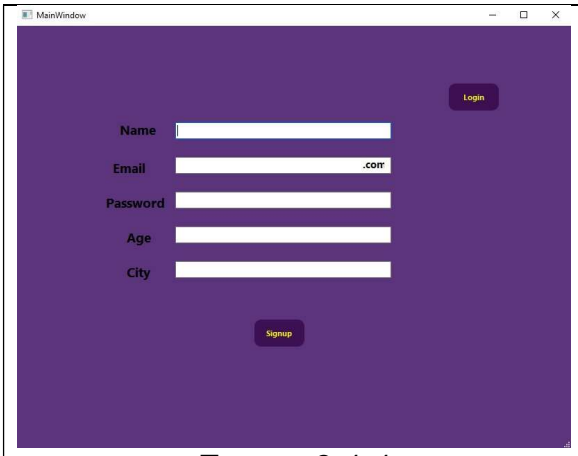
Frame 4



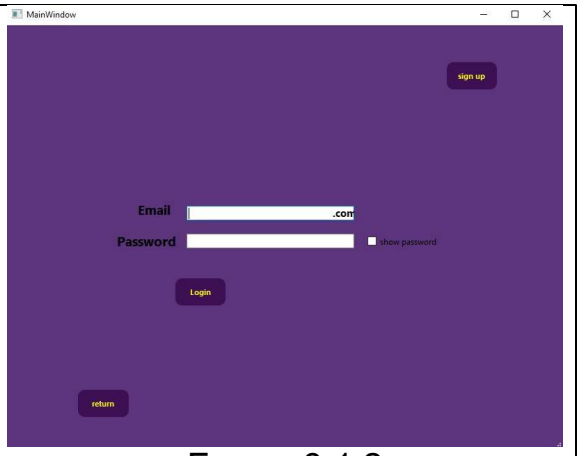
Frame 5



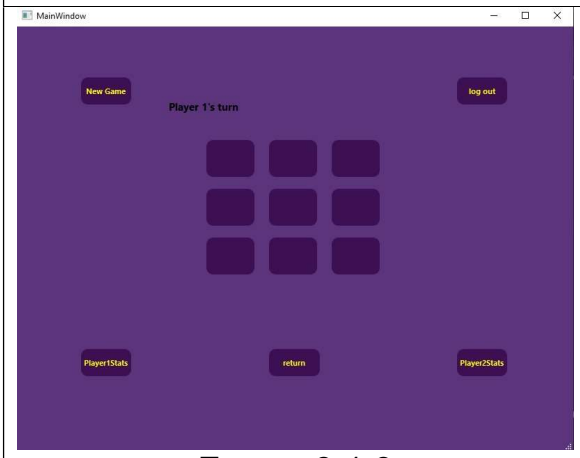
Frame 6.1



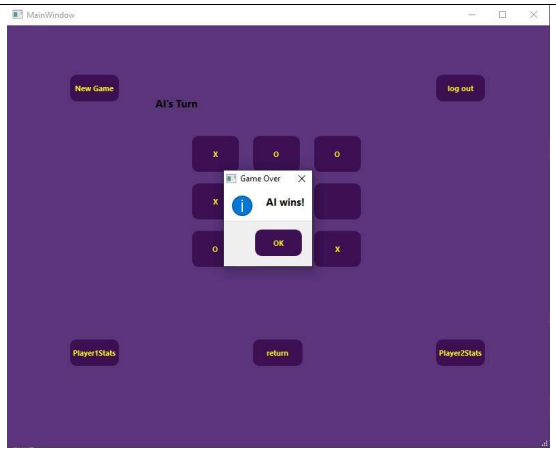
Frame 6.1.1



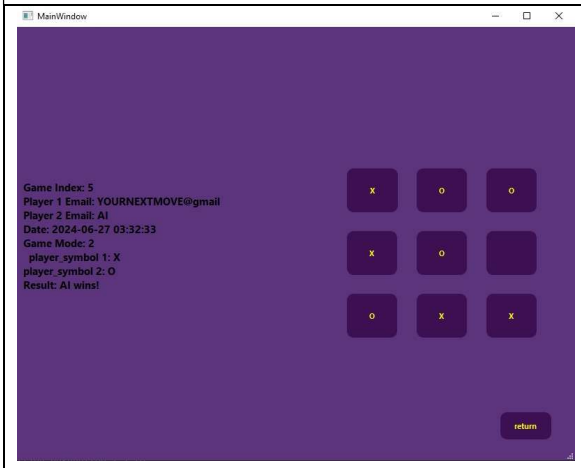
Frame 6.1.2



Frame 6.1.3



Frame 6.2



Frame 7

10.4 Performance Checking Results:

```
07:46:54: Starting J:\qt2\ttt\ttt\build\Desktop_Qt_6_7_0_MinGW_64_bit-Debug\d
Database opened successfully at path: ddd.db
Found table: games
Found table: moves
Found table: players
Tables created successfully or already exist.
"initializeGame()" took 84 milliseconds.
Memory usage: 37000 KB
Kernel CPU time: 343 milliseconds
User CPU time: 156 milliseconds
24 X-------- X 1
98
"handleButtonClick" took 0 milliseconds.
Memory usage: 37216 KB
Kernel CPU time: 390 milliseconds
User CPU time: 156 milliseconds
AI Move:
24 X---O---- O 2
99
"makeAIMove()" took 0 milliseconds.
Memory usage: 43740 KB
Kernel CPU time: 421 milliseconds
User CPU time: 187 milliseconds
24 X---O-X-- X 3
100
"handleButtonClick" took 0 milliseconds.
Memory usage: 43716 KB
Kernel CPU time: 421 milliseconds
User CPU time: 203 milliseconds
AI Move:
24 X--OO-X-- O 4
101
"makeAIMove()" took 0 milliseconds.
Memory usage: 43836 KB
Kernel CPU time: 453 milliseconds
User CPU time: 203 milliseconds
24 X--OOXX-- X 5
102
```

```
102
"handleButtonClick" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 484 milliseconds
User CPU time: 218 milliseconds
AI Move:
24 XO-OOXX-- O 6
103
"makeAIMove()" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 500 milliseconds
User CPU time: 218 milliseconds
24 XO-OOXXX- X 7
104
"handleButtonClick" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
AI Move:
24 XO-OOXXXO O 8
105
"makeAIMove()" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
24 XOXOOXXXO X 9
106
1
"handleGameOutcome(const std::string& player1Email, const std::string& player2Email, int res )" took 91 milliseconds.
Memory usage: 43856 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
Updating board: XOXOOXXXO
Setting pushButton 0 0 to X
```

```
24 XO-OOXXX- X 7
104
"handleButtonClick" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
AI Move:
24 XO-OOXXXO O 8
105
"makeAIMove()" took 0 milliseconds.
Memory usage: 43848 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
24 XOXOOXXXO X 9
106
1
"handleGameOutcome(const std::string& player1Email, const std::string& player2Email, int res )" took 91 milliseconds.
Memory usage: 43856 KB
Kernel CPU time: 515 milliseconds
User CPU time: 218 milliseconds
Updating board: XOXOOXXXO
Setting pushButton_0_0 to X
Setting pushButton_0_1 to O
Setting pushButton_0_2 to X
Setting pushButton_1_0 to O
Setting pushButton_1_1 to O
Setting pushButton_1_2 to X
Setting pushButton_2_0 to X
Setting pushButton_2_1 to X
Setting pushButton_2_2 to O
"showNextMove()" took 5120 milliseconds.
Memory usage: 45436 KB
Kernel CPU time: 890 milliseconds
User CPU time: 421 milliseconds
Total execution time: 34210 ms
Initial memory usage: 16334848 bytes
Final memory usage: 46612480 bytes
07:47:28: J:\qt2\ttt\ttt\build\Desktop_Qt_6_7_0_MinGW_64_bit-Debug\debug\ttt.exe exited with code 0
```