Cairo University, Faculty of Engineering

Electronics and Electrical Communications

Department (EECE)

**Tic-Tac-Toe**

**Bafana Bafana Team**

|  |  |
| --- | --- |
| Name | ID |
| عبد الرحمن احمد محمد عبد اللطيف | 9220457 |
| اسلام فتحي محمد سليمان | 9220145 |
| عبد الرحمن عمرو صلاح الدين | 9220468 |
| سالم جمال عوض الله | 9210462 |
| عمر خالد عبدالعال محمد | 9220535 |

1. **Software Requirements Specifications (SRS)**

## **Functional Requirements**

* This section describes the game rules for Tic-Tac-Toe.
* Tic-Tac-Toe is a game played by two players. One player uses the symbol "X" and the other uses the symbol "O".
* The game is played on a 3x3 grid. Players take turns placing their symbols in one of the nine available cells. The objective is to be the first player to align three of their symbols vertically, horizontally, or diagonally. If neither player achieves this, the game ends in a draw.
* The system must support both player-vs-player and player-vs-AI modes.
* Players must be able to log in, manage their profiles, and track their game history.
* The AI opponent must make strategic moves using advanced algorithms such as minimax with alpha-beta pruning.
* The GUI must be intuitive and user-friendly, allowing for easy interaction with the game and user management features.

## **Nonfunctional Requirements**

This section describes the performance and system behavior requirements.

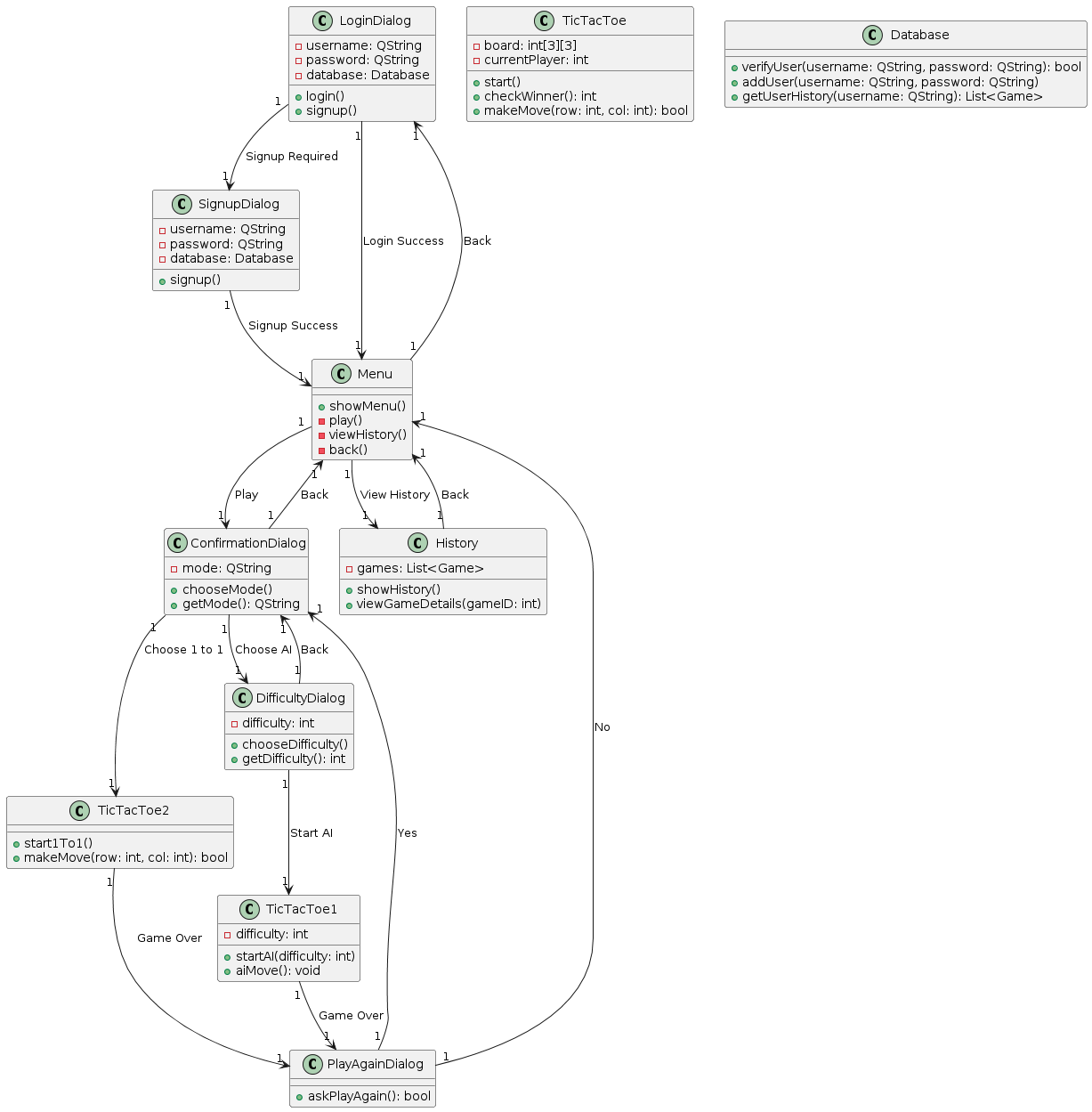
* The system must ensure secure user authentication and data storage using best practices in password hashing and session management.
* The game must provide a responsive and smooth user experience, with minimal latency in gameplay and UI interactions.
* The system must be robust and reliable, with comprehensive unit and integration tests to ensure functionality.
* Continuous integration and deployment practices must be implemented to maintain high code quality and facilitate smooth updates.
* The game must be optimized for performance, with regular monitoring and improvements to response time and resource utilization.

1. **Software Design Specification (SDS):**

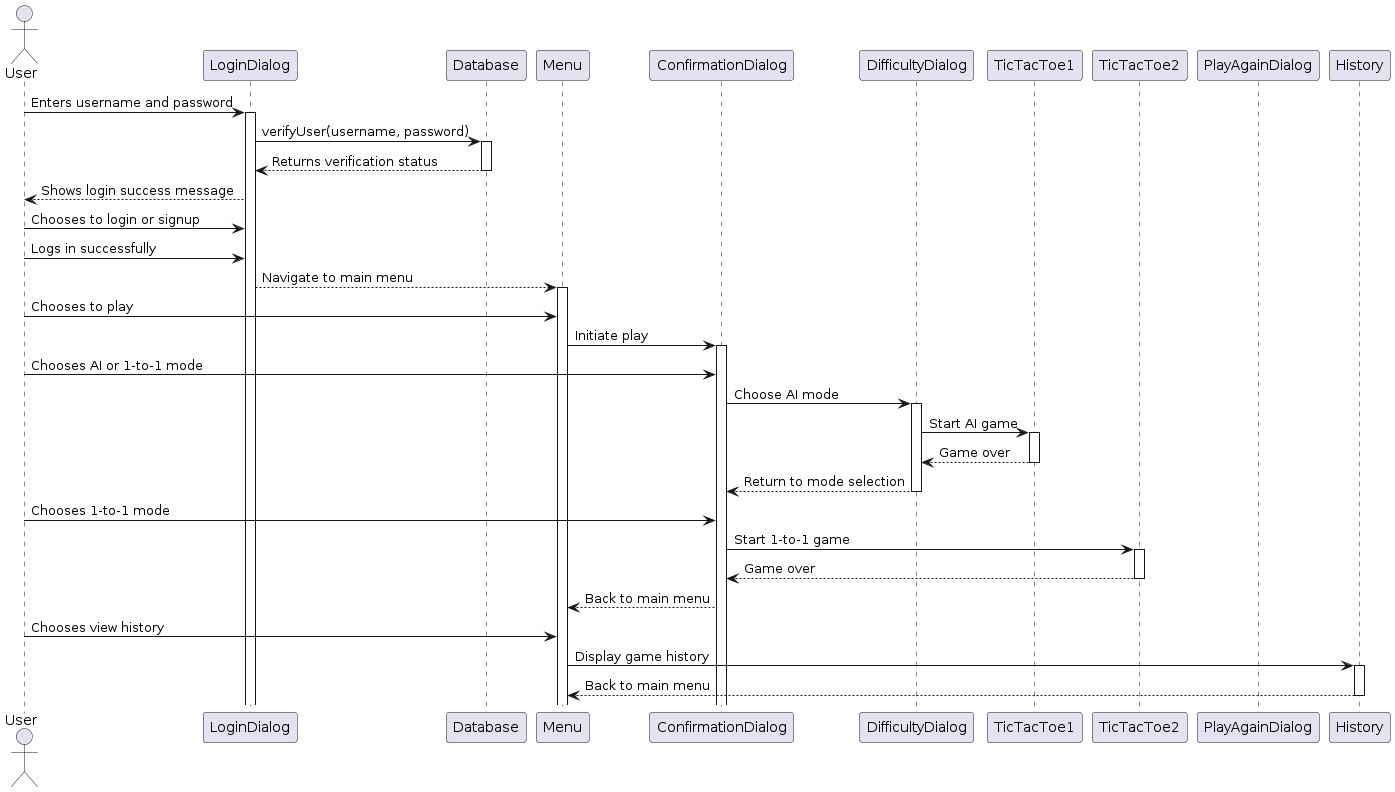
## **Classes and Their Responsibilities**

* LoginDialog Class
  + Methods:
  + login(): Handles the login process.
  + signup(): Redirects to the signup process if the user needs to create a new account.
  + Attributes:
  + username: Stores the entered username.
  + password: Stores the entered password.
  + database: A reference to the `Database` class for user verification.
* -Interactions:
  + Upon successful login, navigates to Menu.
  + If the signup process is required, navigates to SignupDialog.
  + If the password is wrong , a message appears and tell the user that the password is wrong .
* SignupDialog Class
  + Methods:
  + signup(): Handles the signup process.
  + Attributes:
  + username: Stores the entered username.
  + password: Stores the entered password.
  + database: A reference to the `Database` class for adding new users.
  + Interactions:
  + Upon successful signup, navigates to Menu.
* Menu Class
  + Methods:
  + showMenu(): Displays the main menu options.
  + play(): Navigates to ConfirmationDialog to choose game mode.
  + viewHistory(): Navigates to `History` to view past games.
  + back(): Returns to LoginDialog.
  + Interactions:
  + Navigates to ConfirmationDialog for playing the game.
  + Navigates to History for viewing game history.
  + Returns to LoginDialog when logging out.
* TicTacToe Class
  + Methods:
* - start (): Initializes a new game.
  + checkWinner(): Checks for a winner.
  + makeMove(row, col): Makes a move on the game board.
  + Attributes:
  + board: Represents the game board.
  + currentPlayer: Keeps track of the current player.
  + Interactions:
  + Base class for game logic, extended by `TicTacToe1` and `TicTacToe2`.
* TicTacToe1 Class
  + Methods:
  + `startAI(difficulty)`: Starts a game against the AI with the specified difficulty.
  + `aiMove()`: Executes a move by the AI.
  + Attributes:
  + `difficulty`: Stores the difficulty level for the AI.
  + Interactions:
  + After the game ends, navigates to `PlayAgainDialog`.
* TicTacToe2 Class
  + Methods:
  + `start1To1()`: Starts a game between two players.
  + `makeMove(row, col)`: Makes a move in the 1-to-1 game.
  + Interactions:
  + After the game ends, navigates to `PlayAgainDialog`.
* ConfirmationDialog Class
  + Methods:
  + `chooseMode()`: Allows the user to choose between AI or 1-to-1 mode.
  + Attributes:
  + `mode`: Stores the selected mode (AI or 1-to-1).
  + Interactions:
  + Depending on the choice, navigates to `DifficultyDialog` for AI or `TicTacToe2` for 1-to-1.
  + Provides an option to return to `Menu`.
* DifficultyDialog Class:
  + Methods:
  + `chooseDifficulty()`: Allows the user to select the difficulty level for the AI.
  + Attributes:
  + `difficulty`: Stores the selected difficulty level.
  + Interactions:
  + After selecting a difficulty level, navigates to `TicTacToe1`.
  + Provides an option to return to `ConfirmationDialog`.
* PlayAgainDialog Class:
  + Methods:
  + `askPlayAgain()`: Asks the user if they want to play again.
  + Interactions:
  + If the user chooses "Yes," navigates to `ConfirmationDialog` to choose the game mode again.
  + If the user chooses "No," returns to `Menu`.
* History Class
  + Methods:
  + `showHistory()`: Displays the history of played games.
  + `viewGameDetails(gameID)`: Shows details for a selected game.
  + Attributes:
  + `games`: A list of games played by the user.
  + Interactions:
  + Allows the user to view game history and return to `Menu`.
* Database Class
  + Methods:
  + `verifyUser(username, password)`: Verifies user credentials.
  + `addUser(username, password)`: Adds a new user to the database.
  + `getUserHistory(username)`: Retrieves the game history for a user.
  + Interactions:
  + Interacts with `LoginDialog` for user verification.
  + Interacts with `SignupDialog` to add new users.
  + Provides user history for `History`.
* Navigation Flow
* Login Process:
  + The application starts at `LoginDialog`.
  + Successful login navigates to `Menu`.
  + If signup is needed, navigates to `SignupDialog`.
* Menu Options:
  + From `Menu`, the user can choose to play the game, view history, or log out.
  + Choosing to play navigates to `ConfirmationDialog`.
* Choosing Game Mode:
  + In `ConfirmationDialog`, the user chooses between AI and 1-to-1 mode.
  + Choosing AI navigates to `DifficultyDialog` to select the difficulty level.
  + Choosing 1-to-1 mode navigates directly to `TicTacToe2`.
* Playing the Game:
  + `TicTacToe1` starts the game against the AI after selecting difficulty in `DifficultyDialog`.
  + `TicTacToe2` starts the 1-to-1 game.
  + After the game ends, navigates to `PlayAgainDialog`.
* Play Again or Exit:
  + In `PlayAgainDialog`, the user chooses whether to play again or return to the menu.
  + Choosing "Yes" navigates to `ConfirmationDialog` to choose the game mode again.
  + Choosing "No" returns to `Menu`.
* Viewing History:
  + From `Menu`, the user can navigate to `History` to view past games.
  + In `History`, the user can view details of a selected game and then return to Menu .

## **Class diagram:**



**Sequence diagram:**

****

# **Testing:**

A thorough summary of the testing procedures carried out to guarantee the Advanced Tic Tac Toe Game's functionality, dependability, and performance can be found in the testing documentation. This section includes reading recommendations to help you better understand the testing process, as well as the goal of the testing documentation and its target audience.

**Testing Strategies:**

* Unit Testing

The Advanced Tic Tac Toe Game's testing manual uses unit testing, a basic testing technique. It entails testing distinct software modules or components separately to guarantee their accuracy and operation. Unit testing in the context of the game focuses on verifying the Player class, AI algorithms, Game class, and other important parts.

* Integration Testing

To verify how well the various Advanced Tic Tac Toe Game modules interact and integrate, integration testing is crucial. Its main goal is to guarantee that all the parts function as a unit and that information moves between them appropriately.

* Tool: Integration tests are performed using the Qt Test framework and GitHub action as

(.yaml) file.

* Purpose:

To verify the functionality of the LoginDialog class, including UI elements, login attempts, and database connection.

* Test Cases:

Verify UI Elements

Purpose: Ensure all necessary UI elements are present.

Result: Passed. All UI elements (titleLabel, usernameEdit, passwordEdit, loginButton, registerButton) are present.

* Verify Successful Login:
  + Purpose: Ensure a valid user can successfully log in.
  + Result: Passed. The valid user "testuser" with password "testpass" successfully logged in, returning QDialog::Accepted.
* Verify Unsuccessful Login:
* Purpose: Ensure an invalid user cannot log in.
* Result: Passed. The invalid user "wronguser" with password "wrongpass" failed to log in, returning QDialog: Rejected.
* Verify Database Connection:
* Purpose: Ensure the test database connection is correctly established.
* Result: Passed. The test database connection is open and functioning correctly.

**3.2 TestHistory :**

* **Purpose:**

To verify the functionality of the History class, including initialization, database operations, and UI elements.

* **Test Cases:**

Test Initialization

* Purpose: Ensure that the History window initializes correctly.
* Procedure: Verify the window title, size, and stylesheet.
* Result: Passed. The window title is "User History", size is 600x600, and the stylesheet is correct.
* **Test Database Operations**

Purpose: Ensure that game data is correctly added to and retrieved from the database.

Procedure: Add a game record and verify the stored values.

* **Result:** Passed. The game data was correctly added and retrieved, matching the expected values.

**Test UI Elements:**

**Purpose:** Ensure that all necessary UI elements are present and correctly initialized.

Procedure: Verify the presence and properties of the title label, back button, and table view.

**Result:** Passed. All UI elements are present and correctly initialized, with the table view showing one row.

* **Test Cases:**
  + - Test Reset Game
    - Purpose: Ensure that the game board resets correctly.
    - Procedure: Reset the game and check the board state.
    - Result: Passed. The board state resets correctly to an empty state.
* **Test Set/Get Board State**
* Purpose: Ensure that the board state can be set and retrieved correctly.
* Procedure: Set a specific board state and retrieve it to check correctness.
* Result: Passed. The board state is set and retrieved correctly.

**Test Cell Click:**

* Purpose: Ensure that clicking a cell updates the board correctly.
* Procedure: Reset the game, simulate a cell click, and check the board state and button text.
* Result: Passed. The cell click updates the board and button text correctly to 'X'.

**TestPlayAgainDialog:**

**Test Cases:**

* Test Yes Button Click
* Purpose: Ensure that clicking the 'Yes' button emits the correct signal.
* Procedure:
* Create an instance of PlayAgainDialog.
* Find the 'Yes' button.
* Simulate a mouse click on the 'Yes' button.
* Verify that the Yes signal is emitted exactly once.
* Result: Passed. Clicking the 'Yes' button emitted the Yes signal exactly once.

**Test No Button Click**

* Purpose: Ensure that clicking the 'No' button emits the correct signal.
* Procedure:
* Create an instance of PlayAgainDialog.
* Find the 'No' button.
* Simulate a mouse click on the 'No' button.
* Verify that the No signal is emitted exactly once.
* Result: Passed. Clicking the 'No' button emitted the No signal exactly once.

# **6. Performance Measurement and Optimization:**

**Performance Metrics:**

* **Response Time:** The average response time ranges from approximately 62 ms to 2187 ms across different operations.
* **Memory Usage:** The memory usage varies significantly during gameplay, ranging from 31,272,960 bytes to 61,610,752 bytes.
* **CPU Utilization:** CPU time peaks at 2187 ms, indicating the maximum load during processing.

## **Optimization Efforts:**

1. **Memory Optimization:** The game shows fluctuations in memory usage. Consider optimizing resource allocation and releasing unused memory more efficiently.
2. **CPU Efficiency:** While CPU utilization peaks are observed, optimizing algorithms and reducing computational complexity could help in smoother gameplay.
3. **Error Handling:** Addressing background image loading failures and database connection issues would enhance user experience and reduce unnecessary resource consumption.

## **Recommendations:**

* Implement caching mechanisms for frequently used resources to reduce load times and memory overhead.
* Profile and optimize AI algorithms to improve decision-making speed and reduce CPU load.
* Implement efficient error handling to manage resources more effectively during runtime.