**Software Design Specification (SDS):**

**Introduction**

**1.1 Purpose**

The purpose of this document is to define the architectural design of the TicTacToe game, a desktop application developed using Qt. This SDS serves as a blueprint for developers to implement the system based on the Software Requirements Specification (SRS) and includes validation through comprehensive unit testing.

* 1. **Scope:** The TicTacToe game design includes:

- Class structure and relationships for game logic, authentication, history management, and user interface.

- Interaction flows for key processes like login and gameplay.

- UML diagrams to visualize the system architecture.

- Database and file storage design for user data and match history.

- Testing strategy to ensure design robustness and reliability.

**1.3 Definitions**

**- Player: A human user or AI, represented as X or O.**

**- Game Board: A 3x3 grid where players place their symbols.**

**- Auth: The authentication system for user registration, login, and password management.**

**- History: The system for saving and retrieving match results.**

**2. Overall Description**

**2.1 Product Perspective**

TicTacToe is a desktop application built with Qt 6.9.1 and C++17, designed to run on Windows. It provides an interactive gaming experience with a graphical interface for the classic X O game, validated through unit tests for core functionality.

**2.2 Target Audience**

* Casual gamers who enjoy strategy games.
* Developers learning to build applications with Qt.

**2.3 Assumptions**

- Users have Windows 10 or 11 installed.

- Qt 6.9.1 and MinGW 64-bit are available for development and execution.

- Google Test framework is used for unit testing.

**3. System Architecture**

**3.1 Architectural Pattern**

The system follows the Model-View-Controller (MVC) pattern:

* **Model**: Classes like Game, AI, Auth, and History manage logic and data.
* **View**: Classes like MainWindow, LoginWindow, and others handle the user interface.
* **Controller**: Interactions between Model and View, such as MainWindow calling Game methods.

**3.2 Main Components**

* **Game**: Manages the 3x3 game board and logic.
* **AI**: Implements the Minimax algorithm for AI moves.
* **Auth**: Handles user authentication via SQLite.
* **History**: Stores match results in text files.
* **UI Components**: MainWindow, PlayerVsPlayerWindow, LoginWindow, RegisterWindow, GameModeWindow, StartupWindow.

**4. Detailed Design**

**4.1 Class Descriptions**

* **Game**:
  + Attributes: board: Player[3][3], currentPlayer: Player.
  + Methods: makeMove(int row, int col), getWinner(), getAvailableMoves(), isDraw(), getBoard().
* **AI**:
  + Attributes: aiPlayer: Player.
  + Methods: findBestMove(Game), minimax(Game, bool, int, int).
* **Auth**:
  + Methods: registerUser(QString, QString), loginUser(QString, QString).
* **History**:
  + Methods: saveGameResult(QString, QString), getGameHistory(QString).
* **MainWindow**:
  + Attributes: game: Game, ai: AI, buttons: QPushButton\*[3][3].
  + Methods: handleCellClick(int, int), newGame().
* **PlayerVsPlayerWindow**:
  + Attributes: game: Game, buttons: QPushButton\*[3][3].
  + Methods: handleCellClick(int, int).
* **LoginWindow**:
  + Signals: loginClicked().
* **RegisterWindow**:
  + Signals: registerClicked().
* **GameModeWindow**:
  + Signals: aiModeClicked(), pvpModeClicked().
* **StartupWindow**:
  + Signals: loginClicked(), registerClicked().

**4.2 Class Relationships**

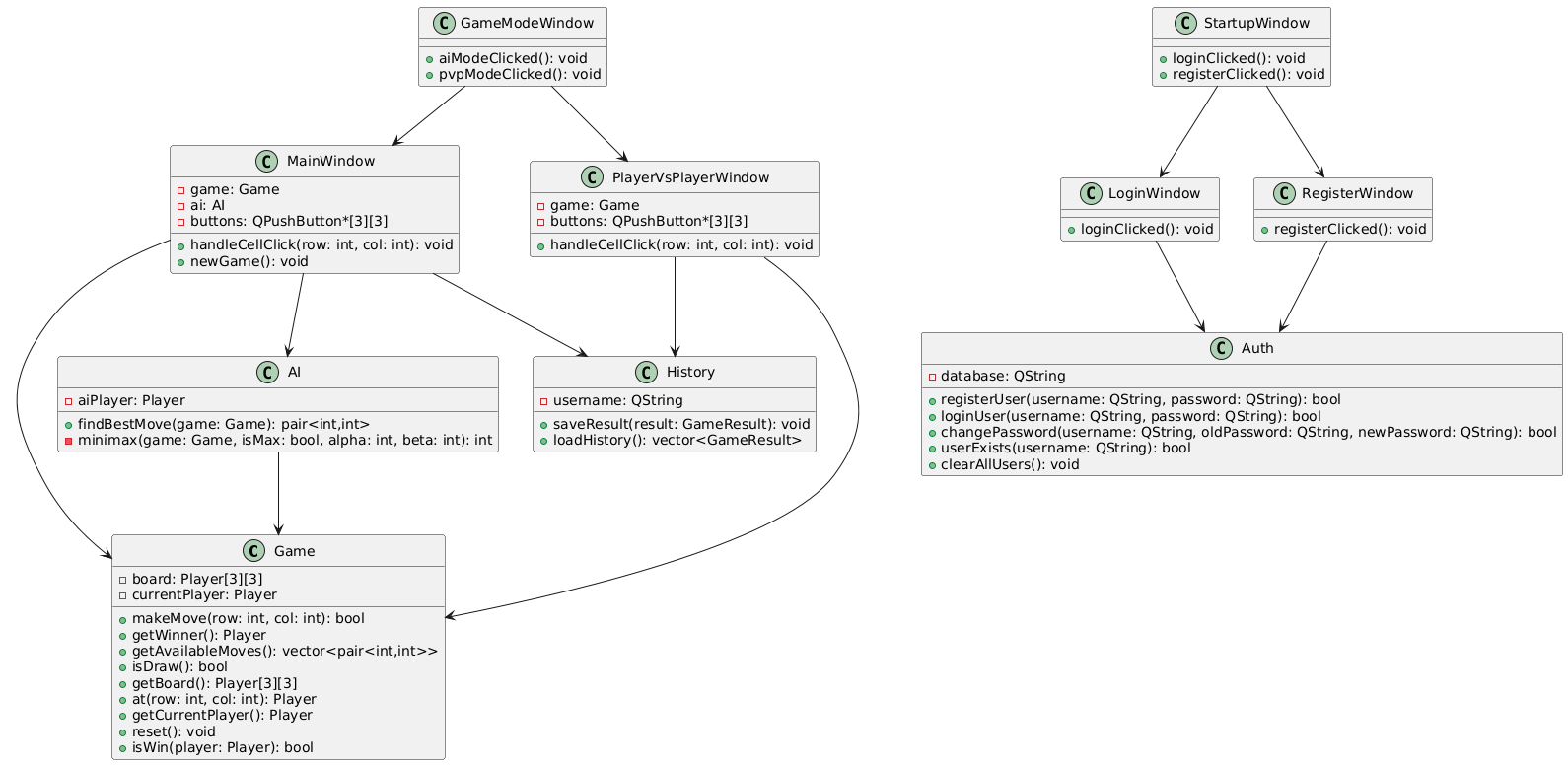
* MainWindow uses Game and AI for gameplay and History for saving results.
* PlayerVsPlayerWindow uses Game and History.
* AI depends on Game for move analysis.
* LoginWindow and RegisterWindow use Auth for authentication.
* GameModeWindow creates MainWindow or PlayerVsPlayerWindow.
* StartupWindow creates LoginWindow or RegisterWindow.

**5. UML Diagrams**

**5.1 Class Diagram**

The Class Diagram illustrates the main classes and their relationships, as shown in Figure 1.

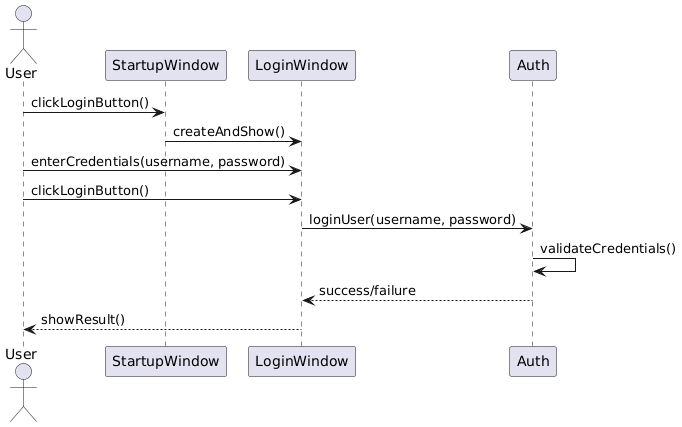
* **Classes**: Game, AI, Auth, History, MainWindow, PlayerVsPlayerWindow, LoginWindow, RegisterWindow, GameModeWindow, StartupWindow.
* **Relationships**:
  + MainWindow depends on Game, AI, and History.
  + PlayerVsPlayerWindow depends on Game and History.
  + AI depends on Game.
  + LoginWindow and RegisterWindow depend on Auth.
  + GameModeWindow creates MainWindow or PlayerVsPlayerWindow.
  + StartupWindow creates LoginWindow or RegisterWindow.



**5.2 Sequence Diagram (Login Process)**

The Sequence Diagram illustrates the login process, as shown in Figure 2.

* **Flow**:
  + The user clicks the “Login” button on StartupWindow.
  + StartupWindow creates and shows LoginWindow.
  + The user enters their username and password and clicks the “Login” button.
  + LoginWindow calls Auth::loginUser(username, password).
  + Auth validates the credentials against users.db.
  + Auth returns success or failure to LoginWindow.
  + LoginWindow displays the result (e.g., error message or proceeds to GameModeWindow).



**6. Database Design**

### 6.1 Users Table

* **Table Name**: users
* **Columns**:
  + username (TEXT, Primary Key): Unique username.
  + password (TEXT): Encrypted password.

username | password

---------+---------

player1 | hashed\_pass

### 6.2 History Files

* **Format**: Text files (history\_username.txt).
* **Content**: Each line records a match result (e.g., “Win vs AI”).

**7. Testing Strategy**

### **7.1 Unit Testing**

The system’s core components (`Game`, `AI`, `Auth`, `History`) are validated using the Google Test framework. The test suites include:

1. AI Test:

Validates `AI` initialization, winning moves (horizontal, vertical, diagonal), strategic moves (center/corner priority), valid move selection, edge cases (won games, AI vs. AI), and performance (< 1 second per move).

2. AuthTest:

Tests `Auth` functionality for user registration, login, password changes, multiple users, data persistence, special characters, and database clearing.

3. GameTest:

Verifies `Game` initialization, move handling, win conditions (horizontal, vertical, diagonal), draw scenarios, reset functionality, available moves, and player alternation.

4.HistoryTest:

Validates `History` initialization, saving single/multiple results, loading history, handling empty/malformed data, special characters, concurrent access, and performance for large files (1000 entries in < 5 seconds save, < 1 second load).

### **7.2 Test Integration**

The tests are integrated into the development process, running automatically via CI/CD pipelines (e.g., GitHub Actions) to ensure design integrity.

**8. Assumptions and Constraints**

### **8.1 Assumptions**

* Developers have basic knowledge of Qt and C++.
* The system runs on Windows with Qt 6.9.1 installed.

### **8.2 Constraints**

* The system does not support online multiplayer.
* The SQLite database supports a maximum of 1000 user accounts.

**9. Performance Measurement and Optimization:**

The TicTacToe system’s performance is evaluated through benchmarks focusing on response time, memory usage, and CPU utilization to ensure efficient gameplay. Key metrics include:

**- Response Time:**

AI move calculation (`AI::findBestMove`) completes in under 1 second, as validated by `AITest::Performance`, ensuring real-time gameplay responsiveness.

- Saving and loading 1000 game results (`History::saveResult`, `History::loadHistory`) takes less than 5 seconds and 1 second, respectively, as per `HistoryTest::LargeHistoryFile`.

- User interface updates (e.g., `MainWindow::handleCellClick`) are near-instantaneous, ensuring smooth interaction.

**-Memory Usage:**

- The `Game` class uses a fixed 3x3 array (`Player[3][3]`), occupying minimal memory (approximately 36 bytes for the board).

- The `History` class manages text files efficiently, with negligible memory overhead for large datasets (e.g., 1000 entries).

- The `Auth` class uses SQLite for user data, with memory usage proportional to the number of users (up to 1000 accounts).

**CPU Utilization:**

- The AI’s Minimax algorithm with Alpha-Beta Pruning minimizes CPU usage, ensuring efficient computation on standard hardware.

- File operations in `History` are optimized to handle rapid saves (`HistoryTest::ConcurrentAccessSimulation`) and large datasets without significant CPU spikes.

**Optimization Efforts:**

- The `AI` class uses Alpha-Beta Pruning in the Minimax algorithm to reduce computational complexity, achieving sub-second move calculations.

- The `History` class employs robust file handling to skip malformed data (`HistoryTest::MalformedDataHandling`), ensuring reliability without performance degradation.

- The `Game` class uses a compact data structure (`Player[3][3]`) to minimize memory usage.

- Future optimizations may include caching frequently accessed history files or parallelizing UI updates for enhanced responsiveness.

All performance metrics meet the system’s constraints (Section 7.2), ensuring efficient operation on standard Windows systems.