Cairo University
Faculty of
Engineering

Dept. of Electronics and Electrical Communications

Second Year

Advanced Tic Tac Toe Game

Software Design Specification (SDS)

NAME	CODE
Islam Essam Salah El Deen	**9230220**
Mohamed Hisham Waheed El Deen	**9230819**
Ahmed Ibrahim Sabry	**9230117**
Amr Khaled Ashour	**9230632**
Mahmoud Mohamed Ezzat	**9230843**
Marwa Ibrahim Fathy	**9230856**

1. Introduction:

1.1 Purpose

This Software Design Specification documents the architectural design and implementation details of the Advanced Tic Tac Toe Game. The system incorporates user authentication, AI opponent with minimax algorithm, personalized game history, and a comprehensive GUI interface built with Qt framework.

1.2 Scope

The SDS covers the complete software architecture including component interactions, database design, user interface specifications, and algorithmic implementations for a multi-modal Tic Tac Toe game supporting both Player vs Player and Player vs AI modes.

1.3 System Overview

The system follows a modular architecture with five primary components:

- **Authentication System**: User registration, login, and session management
- **Game Logic Engine**: Board management and AI opponent implementation
- **Game History Management**: Persistent storage and retrieval of game data
- **Graphical User Interface**: Qt-based multi-window application
- **Testing Framework**: Comprehensive unit and integration testing

---

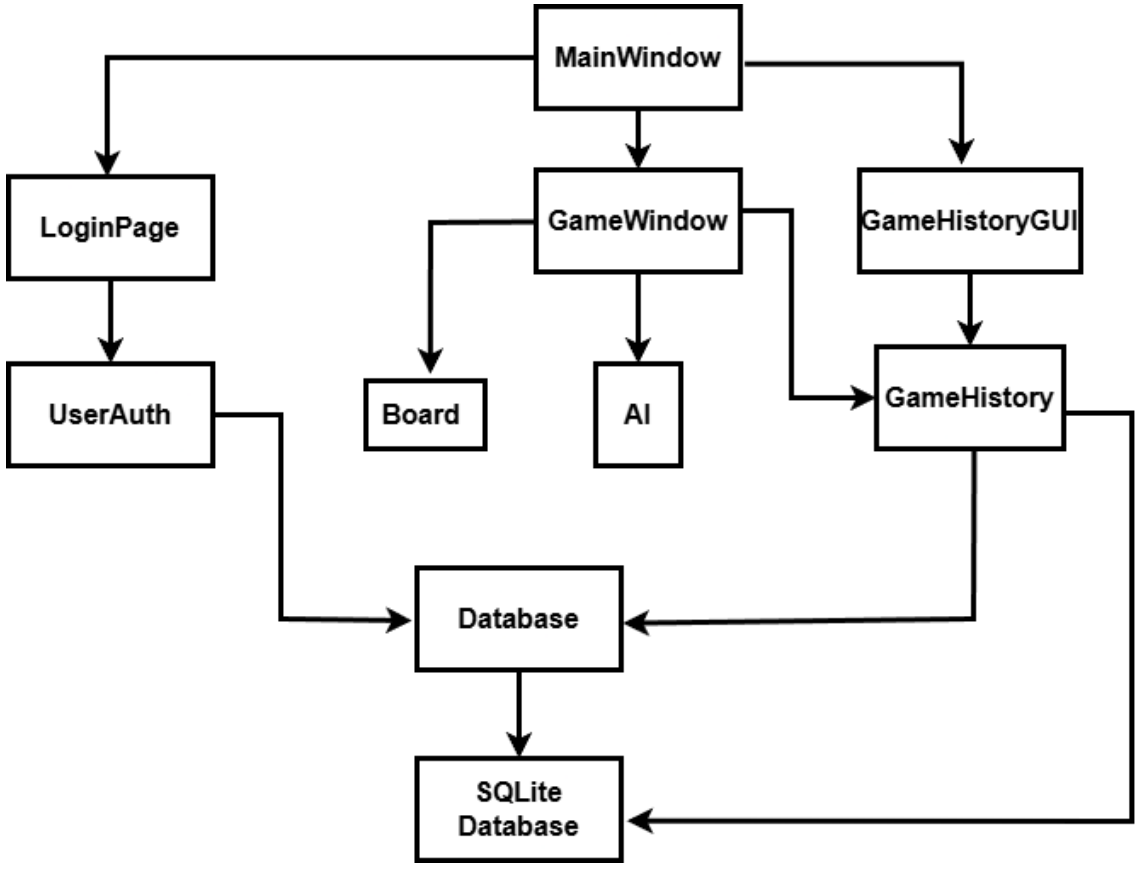
2. System Architecture:

2.1 High-Level Architecture

The system employs a **Layered Architecture** pattern with clear separation of concerns:

- **Presentation Layer**: Qt GUI components (MainWindow, LoginPage, GameWindow, GameHistoryGUI)
- **Business Logic Layer**: Game engine (Board, AI), Authentication (UserAuth)
- **Data Access Layer**: Database abstraction (Database, GameHistory)
- **Data Storage Layer**: SQLite databases for users and game history

2.2 Component Dependencies



2.3 Design Patterns Implemented

- **Model-View-Controller (MVC)**: Separation between GUI (View), game logic (Model), and user interaction handling (Controller)
- **Strategy Pattern**: AI algorithm implementation allowing for different difficulty levels and strategies
- **Observer Pattern**: Qt signals/slots mechanism for event-driven communication between components
- **Singleton-like Pattern**: Single instances of database connections and game state management

---

3. Detailed Component Design:

3.1 Authentication System

3.1.1 UserAuth Class

```
class UserAuth {
    private:
        Database db;
        std::string hashPassword(const std::string& password);
        bool verifyPassword(const std::string& password, const std::string& storedHash);
        bool isValidPassword(const std::string& password);

    public:
        UserAuth(const std::string& dbPath);
        bool registerUser(const std::string& username, const std::string& password);
        bool login(const std::string& username, const std::string& password);
};
```

- **dbPath:** The file path where user account information is stored
- **username:** The unique identifier for each player account
- **password:** The user's chosen password (gets converted to hash for security)
- **Return values:** true means success, false means failure or error

Design Decisions:

- **Security**: SHA-256 hashing using OpenSSL for password storage
- **Validation**: Password complexity requirements (minimum 5 characters, letters + digits)
- **Database Integration**: Abstracted through Database class for maintainability

3.1.2 Database Class

```
class Database {
    private:
        sqlite3* db;
        bool executeQuery(const std::string& query);
    public:
        Database(const std::string& dbPath);
        ~Database();
        void init();
        bool addUser(const std::string& username, const std::string& hashedPassword);
        bool userExists(const std::string& username);
        bool getUserPassword(const std::string& username, std::string& hashedPassword);
};
```

- **sqlite3* db:** This is SQLite's way of representing a database connection
- **query:** SQL command string like "SELECT * FROM users"
- **hashedPassword:** The encrypted version of the user's password for security

3.2 Game Logic Engine

3.2.1 Board Class Architecture

```
class Board {
    private:
        Player grid[3][3];
    public:
        bool makeMove(int row, int col, Player p);
        bool isValidMove(int row, int col) const;
        bool isCellEmpty(int row, int col) const;
        bool isFull() const;
        WinInfo checkWinner() const;
        bool isGameOver() const;
        void reset();     };
```

- **row, col:** Grid coordinates (0-2 for each, like row 0 col 1 is top-middle)
- **Player p:** Either X or O player making the move
- **WinInfo:** Contains winner information and winning line details

State Management:

- **Grid Representation**: 3x3 array of Player enum values
- **Move Validation**: Boundary checking and cell availability verification
- **Win Detection**: Comprehensive checking for rows, columns, and diagonals
- **Game State**: Full board detection and game termination logic

3.2.2 AI Implementation

```
// Minimax with Alpha-Beta Pruning
int minimax(Board board, Player currentPlayer, Player aiPlayer,
            int alpha, int beta, int depth);

std::pair<int, int> findBestMove(const Board& board, Player aiPlayer);
```

Algorithm Specifications:

- **Minimax Algorithm**: Complete game tree search with alpha-beta pruning
- **Optimization**: Depth-based scoring for faster wins preference
- **First Move Optimization**: Center position selection for empty board
- **Immediate Win Detection**: Priority checking for winning moves
- **Time Complexity**: O(b^d) where b=branching factor, d=depth
- **Space Complexity**: O(d) for recursion stack

3.3 Game History System

3.3.1 GameHistory Class Design

```
class GameHistory : public QObject {
    Q_OBJECT
    struct Move { int position; };
    struct GameRecord {
        int id;
        std::vector<Move> moves;
        std::optional<int> playerX_id;
        std::optional<int> playerO_id;
        std::optional<int> winner_id;
        std::chrono::system_clock::time_point timestamp;
    };
    public:
        int initializeGame(std::optional<int> playerX_id, std::optional<int> playerO_id);
        bool recordMove(int game_id, int position);
        bool setWinner(int game_id, std::optional<int> winner_id);
        std::vector<GameRecord> getPlayerGames(int player_id);
};
```

- **std::optional<int>:** Can hold a number or be empty (used when AI plays, since AI has no user ID)
- **position:** Board positions numbered 0-8 in reading order (top-left to bottom-right)
- **game_id:** Unique identifier assigned to each game session
- **Data Serialization**: JSON-like string format for move sequences
- **Signal Emissions**: Qt signals for real-time UI updates

---

4. User Interface Architecture:

4.1 State Management

4.1.1 MainWindow Controller

Responsibilities:

- Window size management based on current view
- User session state tracking
- Component lifecycle management
- Signal routing between components

4.1.2 GameWindow State Machine

UI States:

1. **Setup UI**: Game mode selection (PvP/PvAI)
2. **Symbol Selection UI**: Player symbol choice for PvP
3. **Game Board UI**: Active gameplay interface
4. **Game Over UI**: Results display and options

4.2 Responsive Design

Window Sizing Strategy:

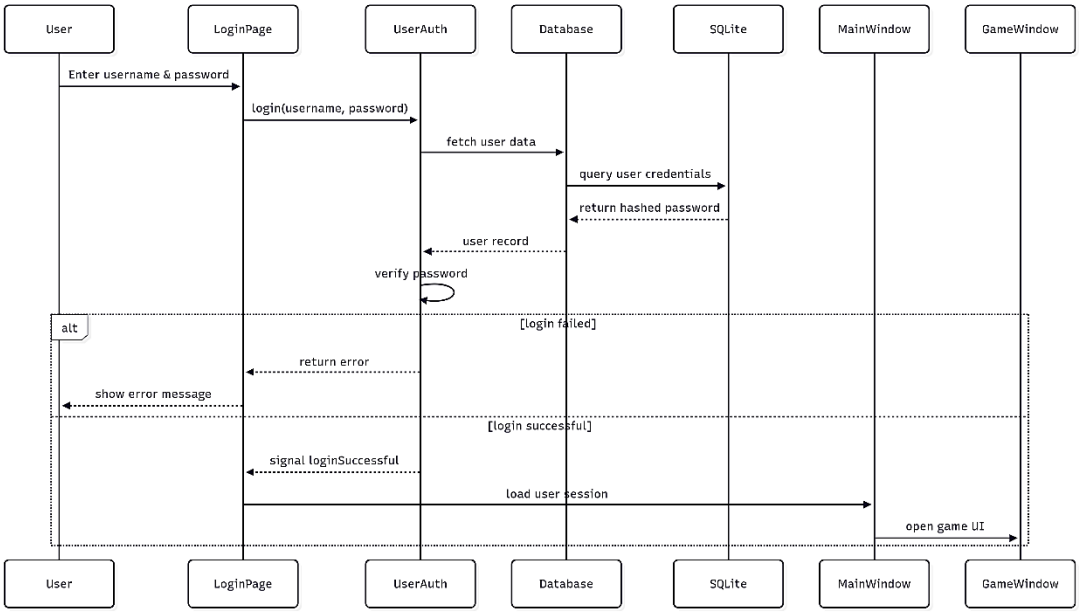
```
namespace UIConstants::WindowSize {
    static constexpr int LOGIN_WIDTH = 600;
    static constexpr int SETUP_WIDTH = 500;
    static constexpr int GAME_WIDTH = 500;
    static constexpr int HISTORY_WIDTH = 1200;
}
```

- **Dynamic Resizing:** Automatic window resizing based on content and user workflow
- **static constexpr:** Values that never change and are set at compile time
- Widths measured in pixels (dots on the screen)
- History window is wider because it displays a table of multiple games

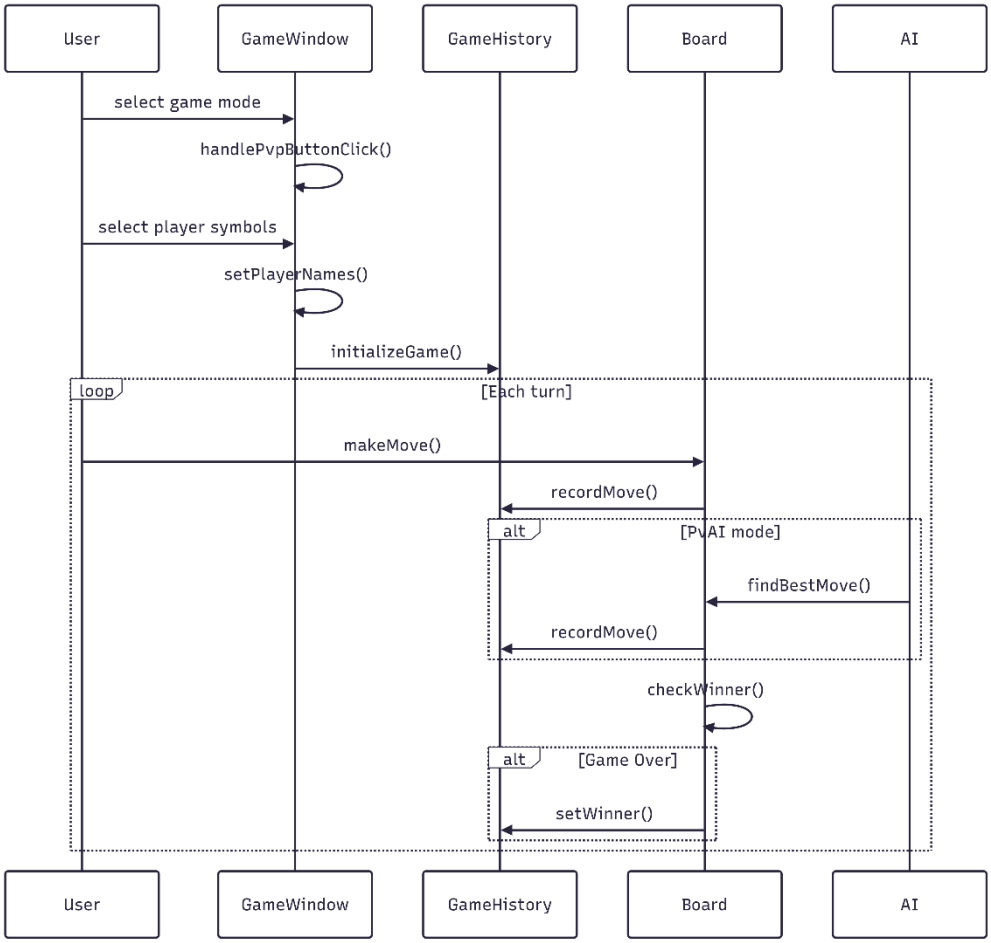
---

5. Data Flow Architecture:

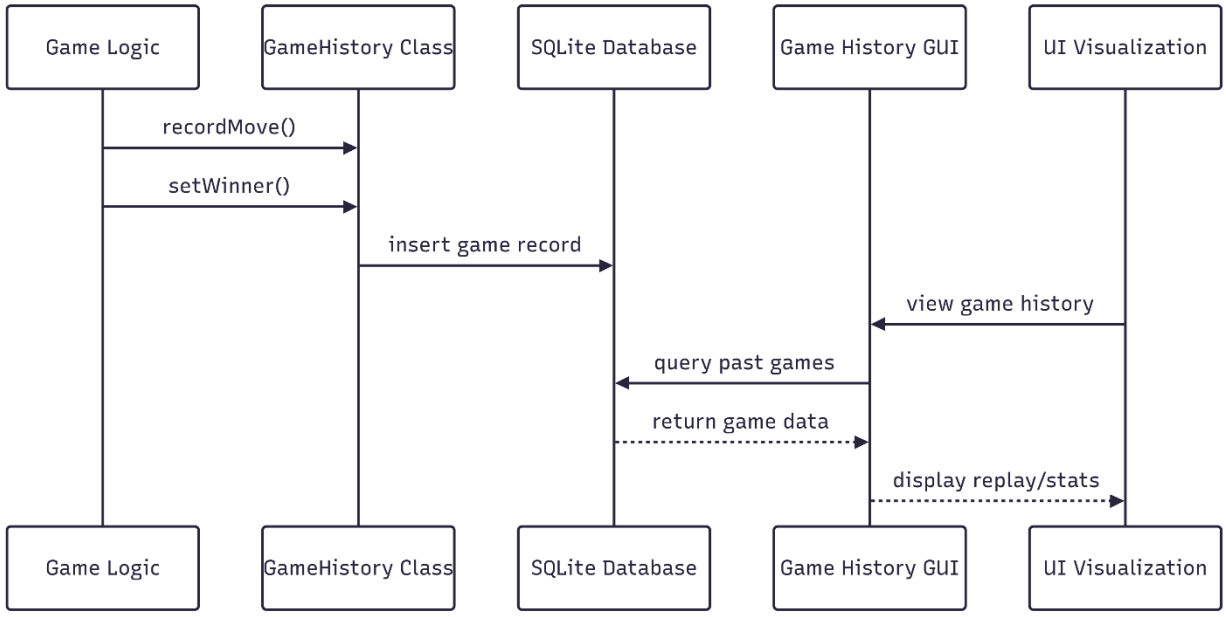
5.1 Authentication Flow



5.2 Game Flow Sequence



5.3 Data Persistence Flow



---

6. Security Architecture:

6.1 Password Security

- **Hashing Algorithm**: SHA-256 with OpenSSL implementation
- **Storage**: No plaintext passwords stored
- **Validation**: Server-side password complexity enforcement

6.2 Database Security

- **SQL Injection Prevention**: Parameterized queries using SQLite prepared statements
- **File Permissions**: Database files with restricted access
- **Session Management**: In-memory user session without persistent tokens

6.3 Input Validation

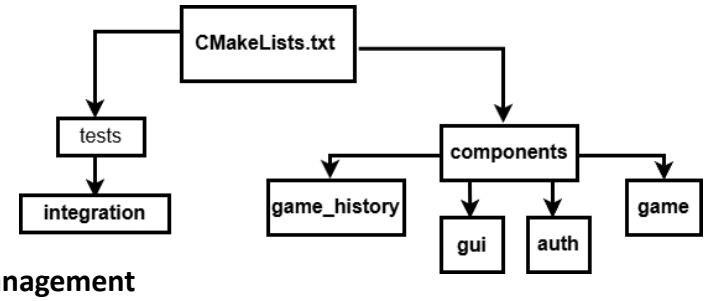
```
bool UserAuth::isValidPassword(const std::string& password) {

    if (password.length() < 5) return false;
    bool hasLetter = false, hasDigit = false;
    for (char c : password) {
        if (std::isalpha(c)) hasLetter = true;
        if (std::isdigit(c)) hasDigit = true;
    }
    return hasLetter && hasDigit;
}
```

- **std::string& password:** Reference to the password string to check
- **std::isalpha(c):** Returns true if character is a letter
- **std::isdigit(c):** Returns true if character is a number
- Function returns true only if password has 5+ characters with both letters and numbers

7. Build System Architecture:

7.1 CMake Structure



7.2 Dependency Management

- **Qt6**: Widgets, Core, Test modules
- **SQLite3**: Database operations
- **OpenSSL**: Cryptographic functions
- **Google Test**: Testing framework

7.3 Cross-Platform Support

- **Windows**: WIN32 executable with resource files
- **Linux/macOS**: Standard executable compilation
- **Resource Management**: Platform-specific asset handling

8.Class Diagram:

9. Conclusion:

This Software Design Specification presents a robust, modular architecture for the Advanced Tic Tac Toe Game. The design emphasizes:

- **Maintainability**: Clear component separation and well-defined interfaces
- **Scalability**: Modular architecture supporting feature extensions
- **Security**: Comprehensive user authentication and data protection
- **Performance**: Optimized algorithms and efficient data structures
- **Testability**: Comprehensive testing framework with high coverage

The implementation successfully integrates modern C++ practices, Qt framework capabilities, and industry-standard security measures to deliver a professional-grade gaming application.