Software Design Specification (SDS)

Advanced Tic Tac Toe Game

Version: 1.0

Date: June 2025

Authors: Amr Khaira,Amr Gamal,Alaa Al-Gazar,Ez-eldeen Negm,Muhammad Haitham

Project: Data Structures Embedded Systems Project - Spring 2025

1. Introduction

1.1 Purpose

This Software Design Specification (SDS) document provides a comprehensive architectural and design overview of the Advanced Tic Tac Toe Game. It details the system architecture, component design, data structures, algorithms, and implementation strategies required to develop a feature-rich Tic Tac Toe application with user authentication, AI opponents, and game history management.

1.2 Scope

The system encompasses a complete Tic Tac Toe gaming platform built in C++ with Qt framework, featuring user management, multiple game modes, AI opponents with varying difficulty levels, comprehensive game history tracking, and a classic graphical user interface with multiple themes.

1.3 System Overview

The Advanced Tic Tac Toe Game is a desktop application that supports both Player vs Player (PvP) and Player vs AI (PvAI) game modes. The system incorporates secure user authentication, personalized game history, intelligent AI opponents using a minimax algorithm, and a sophisticated UI with multiple visual themes.

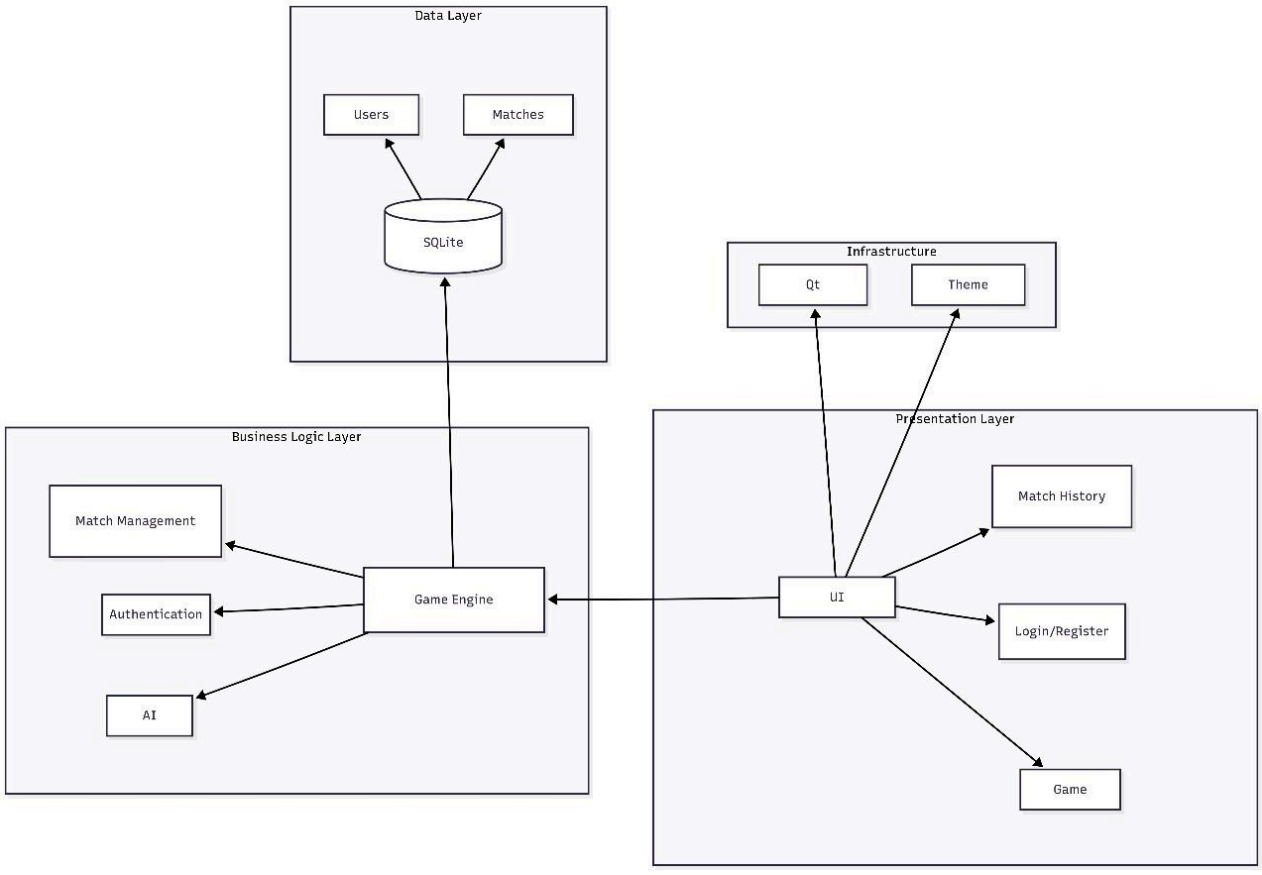
2. System Architecture

2.1 Architectural Pattern

The system follows a **Model-View-Controller (MVC)** architectural pattern with additional layers for data persistence and AI logic:

- **Presentation Layer**: Qt-based GUI components
- **Business Logic Layer**: Game logic, AI algorithms, user management
- **Data Access Layer**: SQLite database operations
- **Persistence Layer**: Database storage for users and match history

2.2 High-Level Architecture



3. Detailed Component Design

3.1 Core Classes

3.1.1 TicTacToe (Main Controller Class)

Purpose: Central controller managing the entire application lifecycle and coordinating between different components.

Key Responsibilities:

- UI management and navigation
- Game state coordination
- User session management
- Database operations coordination

Key Attributes:

```
class TicTacToe : public QMainWindow {
private:
    // Game State
    int mode;                    // 1: PvP, 2: PvAI
    int difficulty;              // 1: Easy, 2: Medium, 3: Hard
    char currentPlayer;          // Current player turn
    std::vector<char> board;     // Game board state
    bool firstMoveMade;          // Track first move

    // User Management
    QString loggedInUser;        // Current logged user
    bool guestMode;              // Guest mode flag

    // Game Series Management
    int totalGames;              // Total games in series
    int gamesToWin;              // Games needed to win series
    int player1Wins, player2Wins, ties;
    QString currentSeriesId;     // Unique series identifier

    // UI Components
    QStackedWidget *stackedWidget;
    QButtonGroup *buttonGroup;
    QLabel *statusLabel, *scoreLabel;
    // ... other UI elements
};
```

3.1.2 Game Logic Components

Board Management:

- 9-element vector representing 3×3 grid
- Character-based representation ('X', 'O', ' ')
- Index mapping: `position = row * 3 + col`

Move Validation:

- Empty cell verification
- Turn-based move enforcement
- Game state validation

3.2 AI Engine Design

3.2.1 AI Difficulty Levels

Easy AI Implementation:

```
int TicTacToe::easyMove() {
    // Priority system:
    // 1. Prefer neutral moves (neither winning nor blocking)
    // 2. Avoid immediate wins when possible
    // 3. Random selection within preferred moves
}
```

Medium AI Implementation:

```
int TicTacToe::mediumMove() {
    // 50% optimal play using minimax
    // 50% worst move selection
    // First move optimization for corners/center
}
```

Hard AI Implementation:

```
int TicTacToe::hardMove() {
    // Full minimax with alpha-beta pruning
```

```
    // Strategic first move (corners when starting, center when responding)
    // Optimal play throughout the game
}
```

3.2.2 Minimax Algorithm

Core Algorithm:

```
int TicTacToe::minimax(std::vector<char> &tempBoard, bool isMaximizing) {
    // Terminal state evaluation
    if (aiWins) return 1;
    if (playerWins) return -1;
    if (tie) return 0;

    // Recursive minimax with alternating players
    int bestScore = isMaximizing ? INT_MIN : INT_MAX;
    for (int i = 0; i < 9; i++) {
        if (tempBoard[i] == EMPTY) {
            tempBoard[i] = isMaximizing ? PLAYER1 : PLAYER2;
            int score = minimax(tempBoard, !isMaximizing);
            tempBoard[i] = EMPTY;
            bestScore = isMaximizing ? max(score, bestScore) : min(score, bestScore);
        }
    }
    return bestScore;
}
```

3.3 User Authentication System

3.3.1 Security Implementation

Password Hashing:

```
QString TicTacToe::pbkdf2Hash(const QString &password, const QByteArray &salt,
                              int iterations, int dkLen) {
    // PBKDF2 implementation with SHA-256
    // 10,000 iterations for security
    // 32-byte derived key length
```

```
}
```

Salt Generation:

```
QByteArray TicTacToe::generateSalt(int length) {
    // Cryptographically secure random salt generation
    // 16-byte salt for password hashing
}
```

3.3.2 Session Management

User States:

- **Authenticated User**: Full access to all features
- **Guest Mode**: Limited access, no history saving
- **Logged Out**: Access only to login/registration

4. Database Design

4.1 Database Schema

4.1.1 Users Table

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    salt TEXT NOT NULL
);
```

4.1.2 Matches Table

```
CREATE TABLE matches (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    player1 TEXT NOT NULL,
    player2 TEXT NOT NULL,
    winner TEXT,
```

```
    result TEXT,
    moves TEXT,
    timestamp TEXT,
    starting_player TEXT,
    game_mode TEXT,
    series_id TEXT,
    game_number INTEGER,
    series_total INTEGER,
    series_target INTEGER
);
```

4.2 Data Access Patterns

Match History Storage:

- Individual game records within series
- Move sequence serialization
- Series tracking with unique identifiers
- Comprehensive metadata storage

Query Optimization:

- Indexed username fields for fast user lookups
- Timestamp-based ordering for history display
- Series-based grouping for replay functionality

5. User Interface Design

5.1 Screen Navigation Flow

```
Login Screen (0) → Mode Selection (1) → Game Settings (5) → Game Screen (2)
     ↓                    ↓                                      ↓
Registration         AI Difficulty (3)                   Match History (6)
     ↓                    ↓                                      ↓
Guest Mode          Player Name (4)                  Recorded Matches (7)
```

5.2 UI Components Architecture

5.2.1 Stacked Widget System

The application uses `QStackedWidget` for screen management with 8 distinct screens:

- **Index 0**: Login/Registration
- **Index 1**: Mode Selection
- **Index 2**: Game Board
- **Index 3**: AI Difficulty Selection
- **Index 4**: Player Name Input
- **Index 5**: Game Settings
- **Index 6**: Match History
- **Index 7**: Recorded Match Replay

5.2.2 Game Board Design

```
// 3x3 Grid Layout with QPushButton array
QGridLayout *gridLayout = new QGridLayout();
QButtonGroup *buttonGroup = new QButtonGroup();
for (int i = 0; i < 9; i++) {
    QPushButton *button = new QPushButton();
    button->setMinimumSize(100, 100);
    gridLayout->addWidget(button, i / 3, i % 3);
    buttonGroup->addButton(button, i);
}
```

5.3 Theme System

5.3.1 Theme Architecture

The application supports multiple visual themes:

- **Light Theme**: Default Windows-style appearance
- **Dark Theme**: Modern dark interface

- **Blue Theme**: Professional blue gradient
- **Specialty Themes**: Plywood, S.P.Q.R, Carthago, Ancient Egypt, etc.

5.3.2 Dynamic Styling

```
void TicTacToe::applyStyleSheet() {
    QString style = generateThemeCSS(selectedTheme);
    setStyleSheet(style);
    updateComponentStyles();
}
```

6. Game Flow and State Management

6.1 Game State Machine

```
[Game Start] → [Player Selection] → [Move Execution] → [Win/Tie Check]
                        ↑                                        ↓
               [Next Player] ← [Continue Game] ← [Game Continues]
                                    ↓
                             [Game End] → [Series Check] → [Series End/Continue]
```

6.2 Series Management

Series Configuration:

- Configurable total games (1-10)
- Configurable games to win (1-10)
- Automatic series progression
- Series winner determination

Game Tracking:

- Individual game results within series
- Move history preservation
- Starting player rotation

7. Replay System Design

7.1 Replay Architecture

Data Storage[1]:

- Move sequence serialization as comma-separated integers
- Starting player preservation
- Game mode identification
- Timestamp-based ordering

Replay Engine:

```
void TicTacToe::replayNextMove() {
    // Calculate current player based on move index and starting player
    char currentReplayPlayer = (replayIndex % 2 == 0) ?
        replayStartingPlayer : (replayStartingPlayer == PLAYER1 ? PLAYER2 : PLAYER1);

    // Execute move and update display
    int moveIndex = replayMoves[replayIndex];
    board[moveIndex] = currentReplayPlayer;
    updateBoard();

    // Check for game end conditions
    checkWinOrTie();
}
```

7.2 Replay Features

Interactive Replay:

- Automatic move progression with timing
- Game state restoration
- Series progression visualization
- Win/loss/tie detection during replay

8. Error Handling and Validation

8.1 Input Validation

User Input Validation:

- Username/password length and character validation
- Move validation (empty cells only)
- Settings bounds checking (games count, win conditions)

Database Error Handling:

- Connection failure recovery
- Query execution error handling
- Data integrity validation

8.2 Game State Validation

Move Validation[1]:

```
void TicTacToe::makeMove(int index) {
    if (board[index] != EMPTY) return;  // Invalid move
    if (mode == 2 && currentPlayer == PLAYER1) return;  // AI turn

    // Execute valid move
    board[index] = currentPlayer;
    moveHistory.push_back(index);
    updateBoard();
}
```

9. Performance Considerations

9.1 Algorithm Optimization

Minimax Optimization:

- Alpha-beta pruning for reduced search space
- Early termination on terminal states
- Efficient board state evaluation

Memory Management:

- Vector-based board representation for efficiency
- Minimal object creation during gameplay
- Efficient string handling for database operations

9.2 Database Performance

Query Optimization:

- Prepared statements for security and performance
- Indexed columns for fast lookups
- Batch operations for series data

10. Security Considerations

10.1 Authentication Security

Password Security[3]:

- PBKDF2 hashing with 10,000 iterations
- Cryptographically secure salt generation
- No plaintext password storage

Session Security:

- Secure session state management
- Automatic logout functionality
- Guest mode isolation

10.2 Data Protection

Database Security:

- SQL injection prevention through prepared statements
- Input sanitization
- Error message sanitization

11. Testing Strategy

11.1 Unit Testing Framework

QT Test Integration[1]:

- Comprehensive unit tests for game logic
- AI algorithm validation
- Database operation testing
- User authentication testing

Test Coverage Areas:

- Game state management
- Win/tie detection algorithms
- AI move generation
- User authentication flows
- Database operations

11.2 Integration Testing

Component Integration:

- UI-to-logic integration testing
- Database connectivity testing
- Theme system validation
- Replay system verification

12. Deployment and Maintenance

12.1 Build System

QMake Configuration:

- Cross-platform build support
- Qt framework integration

- Google Test framework integration
- SQLite dependency management

12.2 Version Control

Git Workflow[1]:

- Feature branch development
- Code review process
- Automated testing integration
- Release tagging strategy

13. Future Enhancements

13.1 Potential Features

Network Multiplayer:

- TCP/IP-based remote play
- Lobby system for player matching
- Real-time game synchronization

Advanced AI:

- Machine learning integration
- Adaptive difficulty adjustment
- Player behavior analysis

Enhanced UI:

- Animation system
- Sound effects integration
- Mobile platform support
- Customizable Themes Generation