Software Design Specification (SDS) for Tic-Tac-Toe

Table of Contents

Definitions, Acronyms, and Abbreviations

AI: Artificial Intelligence

SRS: Software Requirements Specification

UI: User Interface

UX: User Experience

QSS: Qt Style Sheets

Qt: A cross-platform application development framework.

Minimax: An algorithm used in AI for decision-making in two-player games.

Alpha-Beta Pruning: An optimization technique for the Minimax algorithm.

SQL: Structured Query Language (used for database interaction).

EEST: Eastern European Summer Time.

PVP: Player versus Player.

app.pro: QMake project file for the main application.

tests.pro: QMake project file for the test suite.

mainwindow.ui: XML file defining the user interface layout.

MOC: Meta-Object Compiler, a tool used by Qt to handle C++ extensions for signals and slots.

Uic: Qt User Interface Compiler, a tool that processes .ui files to generate C++ header files.

QSettings: A Qt class for storing application settings persistently.

SQLite: A C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.

SHA-256: Secure Hash Algorithm 256-bit, a cryptographic hash function.

QMake: A build system tool used by Qt to manage project files.

SDS: Software Design Specification. This document itself.

UML: Unified Modeling Language. A standard for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.

1. Introduction

1.1. Purpose

This Software Design Specification (SDS) document describes the software architecture and detailed design of the Tic-Tac-Toe application. It outlines the modular structure, component responsibilities, and key interactions, including UML class and sequence diagrams, to guide the implementation and ensure a well-structured and maintainable system.

1.2. Scope

The scope of this SDS is to detail the internal design of the Tic-Tac-Toe application, which provides user authentication, two-player and AI game modes, game history management, and replay functionality. This document translates the requirements specified in the Software Requirements Specification (SRS) into a concrete design.

2. Overall Design

2.1. System Overview

The Tic-Tac-Toe application is designed as a desktop-based game with user management and history tracking capabilities. It employs a layered architecture, separating presentation (UI), business logic (game rules), data access, and auxiliary components. The application leverages the Qt framework for its graphical user interface and event handling, while utilizing SQLite as an embedded database for persistent data storage.

2.2. Design Goals and Principles

The primary design goals for the Tic-Tac-Toe application include modularity, maintainability, testability, and responsiveness. Modularity is achieved through clear separation of concerns, assigning distinct responsibilities to individual classes. This promotes maintainability by localizing changes and simplifies testing by allowing components to be tested independently. Responsiveness is targeted for AI decision-making and database operations to ensure a fluid user experience. The design adheres to common software engineering principles such as low coupling and high cohesion.

3. Architectural Design

3.1. Architectural Style

The application follows a variation of the Model-View-Controller (MVC) architectural pattern, adapted for a Qt desktop application.

- **View (Presentation Layer):** Handled primarily by the `MainWindow` class and the `mainwindow.ui` file, which define the visual layout and user interaction elements. User input events (e.g., button clicks) are captured here.
- **Controller/Logic Layer:** The `MainWindow` acts as a controller, coordinating interactions between the UI and the underlying game logic and data management components. The `GameLogic` class encapsulates the core game rules and state management, effectively serving as a model and an internal controller for the game itself.
- **Model (Data Layer):** The `Board` class represents the current state of the Tic-Tac-Toe board (the game model). The `DatabaseManager` class provides the interface for persistent storage of user accounts and game history, acting as the data access model.

3.2. Module Structure

The project is structured into logical modules, reflected in the directory organization and QMake project files.

- **`app` Module:** Contains the core application logic and UI.
  - `mainwindow.h`, `mainwindow.cpp`: Main application window, UI coordination, and top-level event handling.
  - `gamelogic.h`, `gamelogic.cpp`: Implements the rules and state transitions of the Tic-Tac-Toe game.
  - `aiplayer.h`, `aiplayer.cpp`: Encapsulates the AI logic, including easy (random) and hard (Minimax) difficulties.
  - `board.h`, `board.cpp`: Manages the 3x3 game board state and basic operations.
  - `databasemanager.h`, `databasemanager.cpp`: Handles all interactions with the SQLite database for user and game history persistence.
  - `messagebox.h`, `messagebox.cpp`: Provides a utility for consistent, styled message boxes.
- **`tests` Module:** Contains the unit tests for the core logic components.
  - `tst_testboard.h`, `tst_testboard.cpp`: Tests the `Board` class.
  - `tst_aiplayer.h`, `tst_aiplayer.cpp`: Tests the `AIPlayer` class.
  - `tst_gamelogic.h`, `tst_gamelogic.cpp`: Tests the `GameLogic` class.
  - `tst_databasemanager.h`, `tst_databasemanager.cpp`: Tests the `DatabaseManager` class.

3.3. Major Components

The application is composed of several major, interconnected components:

- **User Interface (UI) Component (`MainWindow`):** Responsible for rendering the application's screens (login, main menu, game board, account, history) and

capturing user input. It mediates between user actions and the underlying business logic.

- **Game Logic Component (`GameLogic`):** Contains the complete ruleset and state management for Tic-Tac-Toe. It processes player moves, determines game outcomes (win/draw), and manages player turns. It communicates with the UI to update the board display and with the AI player for computer turns.
- **AI Component (`AIPlayer`):** Implements the artificial intelligence for the single-player mode. It offers different difficulty levels, with the "Hard" mode utilizing the Minimax algorithm with Alpha-Beta Pruning for optimal play.
- **Data Management Component (`DatabaseManager`):** Provides an abstraction layer for interacting with the SQLite database. It handles user registration, authentication, password resets, and the saving, loading, and deletion of game history records.
- **Game Board Component (`Board`):** Represents the actual 3x3 game board grid, storing the state of each cell. It provides methods for making moves, checking for a full board, and detecting win conditions.

4. Detailed Design

4.1. Class Design

The system's detailed structure is best represented by a class diagram, illustrating the relationships and responsibilities of each primary class.

4.1.1. Class Diagram

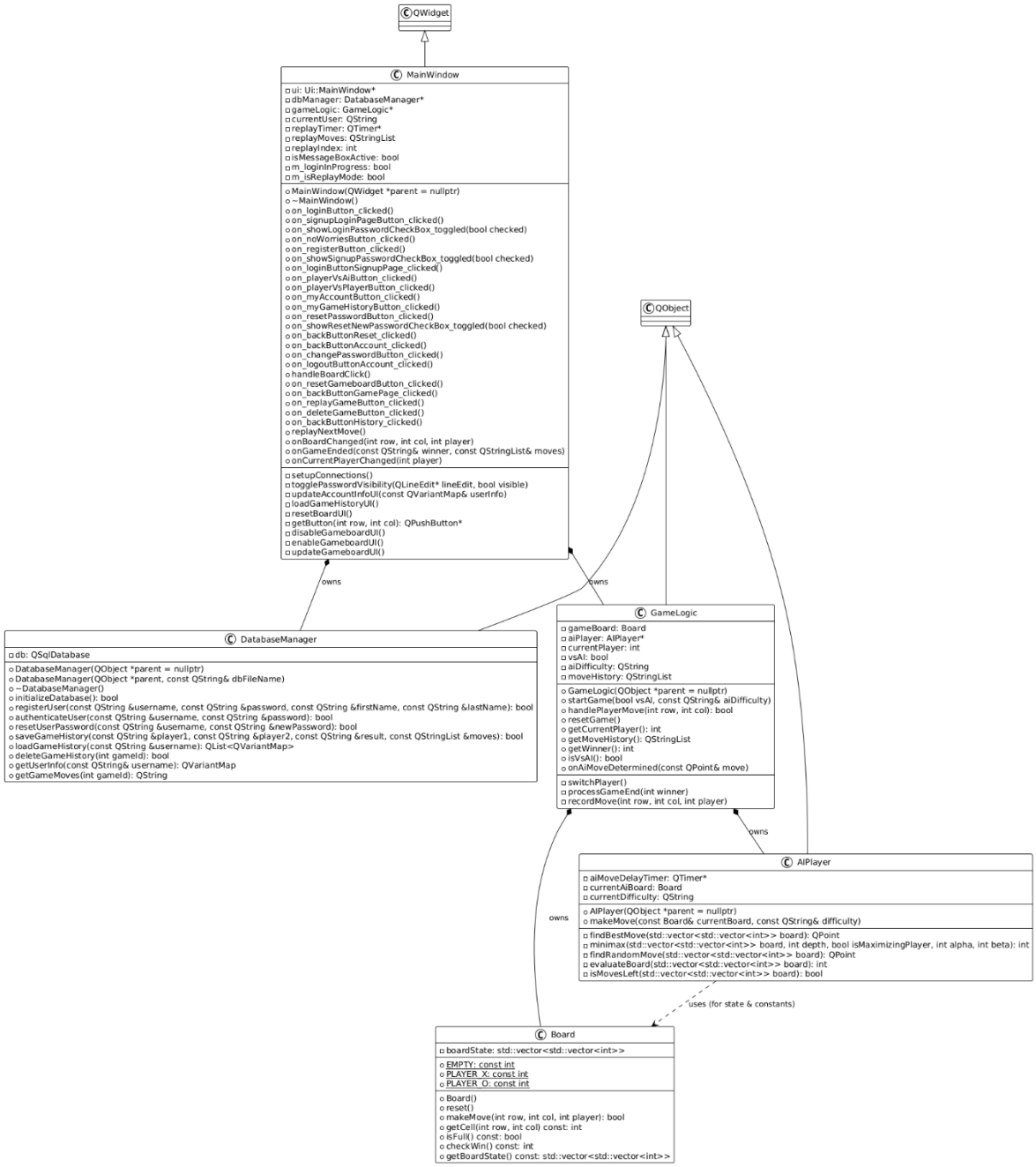

Figure 4.1: Class Diagram of Tic-Tac-Toe Application

4.2. Interaction Design

The system's behavior in response to key user actions is described through sequence diagrams, illustrating the flow of messages between objects.

4.2.1. User Login Sequence

This sequence describes the steps involved when a user attempts to log in, including validation, authentication, and UI updates.

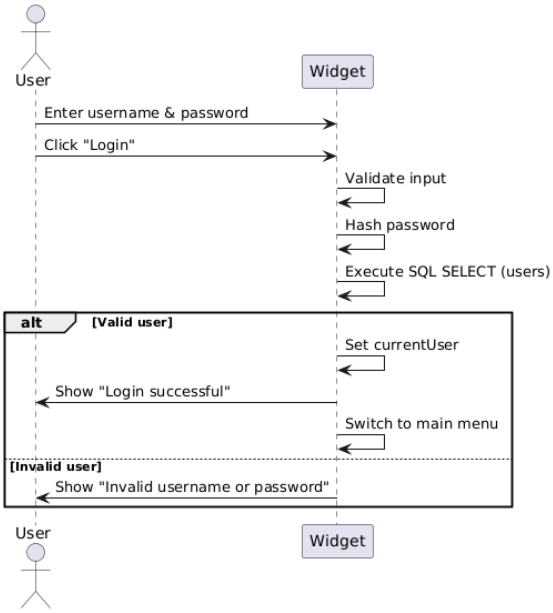

Figure 4.2.1: User Login Sequence Diagram

4.2.2. Player Move Sequence

This sequence details how a human player makes a move on the game board and how the system responds, including checking for game end conditions.

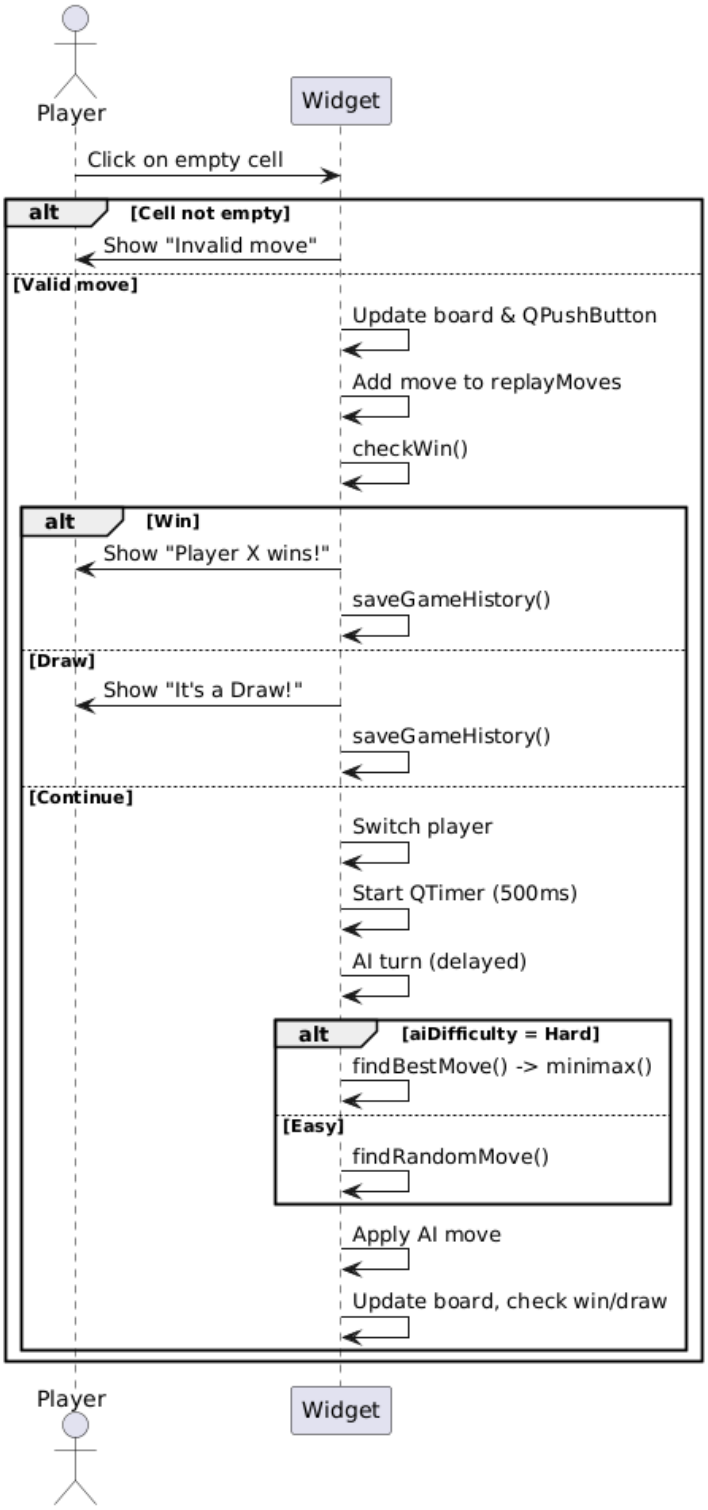

Figure 4.2.2: Player Move & AI Response Sequence Diagram

4.2.3. AI Move Sequence

This sequence illustrates the AI's turn, from being requested to calculating and making a move, and the subsequent system responses.

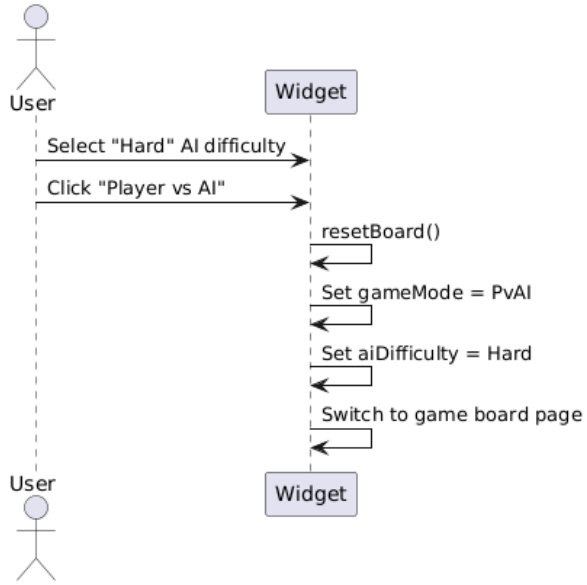

Figure 4.2.3: Start PvAI Game (Hard Difficulty) Sequence Diagram

4.2.4. Game Replay Sequence

This sequence illustrates how a user initiates a game replay from the history, and how the moves are animated on the board.

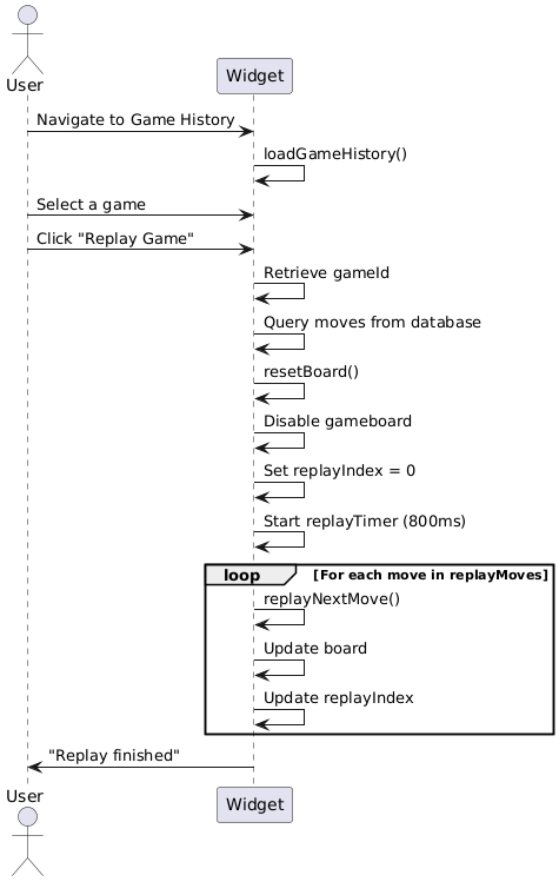

Figure 4.2.4: Replay Game History Sequence Diagram

5. Data Design

5.1. Database Schema

The application utilizes an embedded SQLite database to persist user accounts and game history. The database schema consists of two main tables: `users` and `game_history`.

- **`users` Table:**
  - `id`: INTEGER PRIMARY KEY AUTOINCREMENT (Unique identifier for each user).
  - `username`: TEXT UNIQUE NOT NULL (User's chosen unique username for login).
  - `password`: TEXT NOT NULL (Hashed password using SHA-256).
  - `firstName`: TEXT (Optional first name of the user).
  - `lastName`: TEXT (Optional last name of the user).
- **`game_history` Table:**
  - `id`: INTEGER PRIMARY KEY AUTOINCREMENT (Unique identifier for each game record).

- `player1`: TEXT NOT NULL (Username of Player 1, typically the logged-in user).
- `player2`: TEXT NOT NULL (Username of Player 2, or "AI" if played against the computer).
- `result`: TEXT NOT NULL (The outcome of the game, e.g., "Player X Wins!", "AI wins!", "Draw").
- `moves`: TEXT NOT NULL (A comma-separated string representing the sequence of moves, e.g., "0:0:X,1:1:O,0:1:X").
- `timestamp`: DATETIME DEFAULT CURRENT_TIMESTAMP (Automatically records the date and time the game record was created).

All interactions with this schema are handled exclusively by the `DatabaseManager` class, which uses prepared statements to ensure data integrity and prevent SQL injection vulnerabilities.