Software Design Specification (SDS) for Advanced Tic Tac Toe Game

1. Introduction

1.1 Purpose

This Software Design Specification (SDS) details the architectural and detailed design of the Advanced Tic Tac Toe game. It serves as a blueprint for implementation, ensuring a well-structured, maintainable, and scalable system. This document translates the requirements outlined in the Software Requirements Specification (SRS) into a concrete design, focusing on how the system components will be built and integrated.

1.2 Scope

The scope of this SDS covers the high-level architecture, major components, their responsibilities, interfaces, and interactions. It also delves into the detailed design of key modules such as User Authentication, Game Logic, AI, and GUI, including data structures and algorithms employed.

1.3 Definitions, Acronyms, and Abbreviations

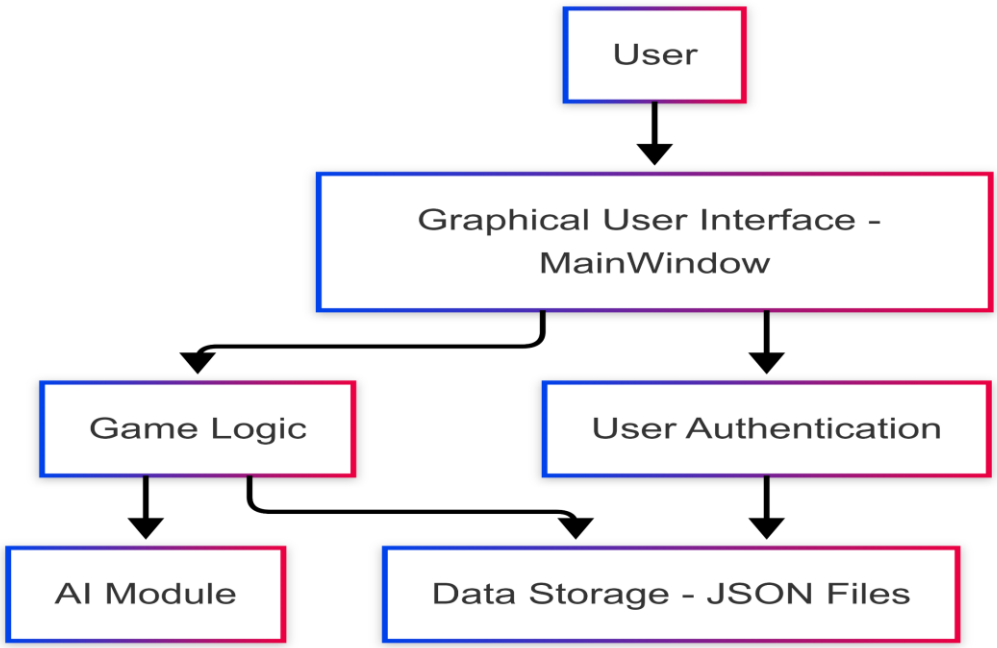
- **AI:** Artificial Intelligence
- **GUI:** Graphical User Interface
- **SRS:** Software Requirements Specification
- **SDS:** Software Design Specification
- **CI/CD:** Continuous Integration/Continuous Deployment
- **Qt:** A cross-platform application development framework.
- **Minimax:** A decision rule used in artificial intelligence, decision theory, game theory, and statistics for minimizing the possible loss for a worst case (maximum loss) scenario.
- **Alpha-Beta Pruning:** A search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree.
- **MVC:** Model-View-Controller, an architectural pattern.

2. System Architecture

2.1 Architectural Style

The system primarily follows a Model-View-Controller (MVC) architectural pattern, particularly evident in the separation of concerns between the `GameLogic` (Model), `MainWindow` (View), and the event handling mechanisms (Controller). The `UserAuth` module acts as a service layer, managing user data and authentication logic, which can be considered part of the Model or a separate service component interacting with the Model.

2.2 High-Level Component Diagram



2.3 Component Descriptions

2.3.1 Graphical User Interface (GUI) - `MainWindow`

- **Responsibility:** Handles all user interactions, displays the game board, user authentication forms, game history, and various game states. It acts as the View and part of the Controller in the MVC pattern.
- **Key Classes:** `MainWindow`
- **Interfaces:** Provides methods for updating the board, displaying messages, handling user input (button clicks, text entry).
- **Dependencies:** Depends on `UserAuth` for authentication services and `GameLogic` for game state and actions.

2.3.2 User Authentication - `UserAuth`

- **Responsibility:** Manages user registration, login, logout, password hashing, and storage/retrieval of user profiles and game histories.
- **Key Classes:** `UserAuth` , `User` (struct/class within `UserAuth` for user data). •
- **Interfaces:** `signUp()` , `signIn()` , `isLoggedIn()` , `getCurrentUser()` , `signOut()` , `saveGameToHistory()` , `getGameHistory()` .
- **Dependencies:** Depends on Qt classes for file I/O (`QFile` , `QJsonDocument` , `QJsonObject` , `QJsonArray`) and cryptographic hashing (`QCryptographicHash`). Interacts with the file system for persistent storage.

2.3.3 Game Logic - `GameLogic`

- **Responsibility:** Implements the core Tic Tac Toe game rules, manages the game board state, current player, win/tie conditions, and move validation. It also orchestrates AI moves and game history recording.
- **Key Classes:** `GameLogic` , `Player` (enum), `Move` (struct).
- **Interfaces:** `newGame()` , `makeMove()` , `checkWin()` , `checkGameOver()` , `getCell()` , `getCurrentPlayer()` , `getWinner()` , `isGameOver()` , `setDifficulty()` , `aiMove()` , `getGameAsJson()` , `loadFromJson()` , `replayMove()` .
- **Dependencies:** Depends on `AI` module for AI moves. Emits signals to `MainWindow` for board updates and game end events.

2.3.4 AI Module

- **Responsibility:** Implements the artificial intelligence for the single-player mode. It uses the Minimax algorithm with Alpha-Beta Pruning to determine optimal moves based on the selected difficulty level.
- **Key Algorithms:** Minimax with Alpha-Beta Pruning.
- **Interfaces:** `minimax()` , `isWin()` , `getAvailableMoves()` , `evaluateBoard()` .
- **Dependencies:** Interacts with `GameLogic` to get the current board state and make moves.

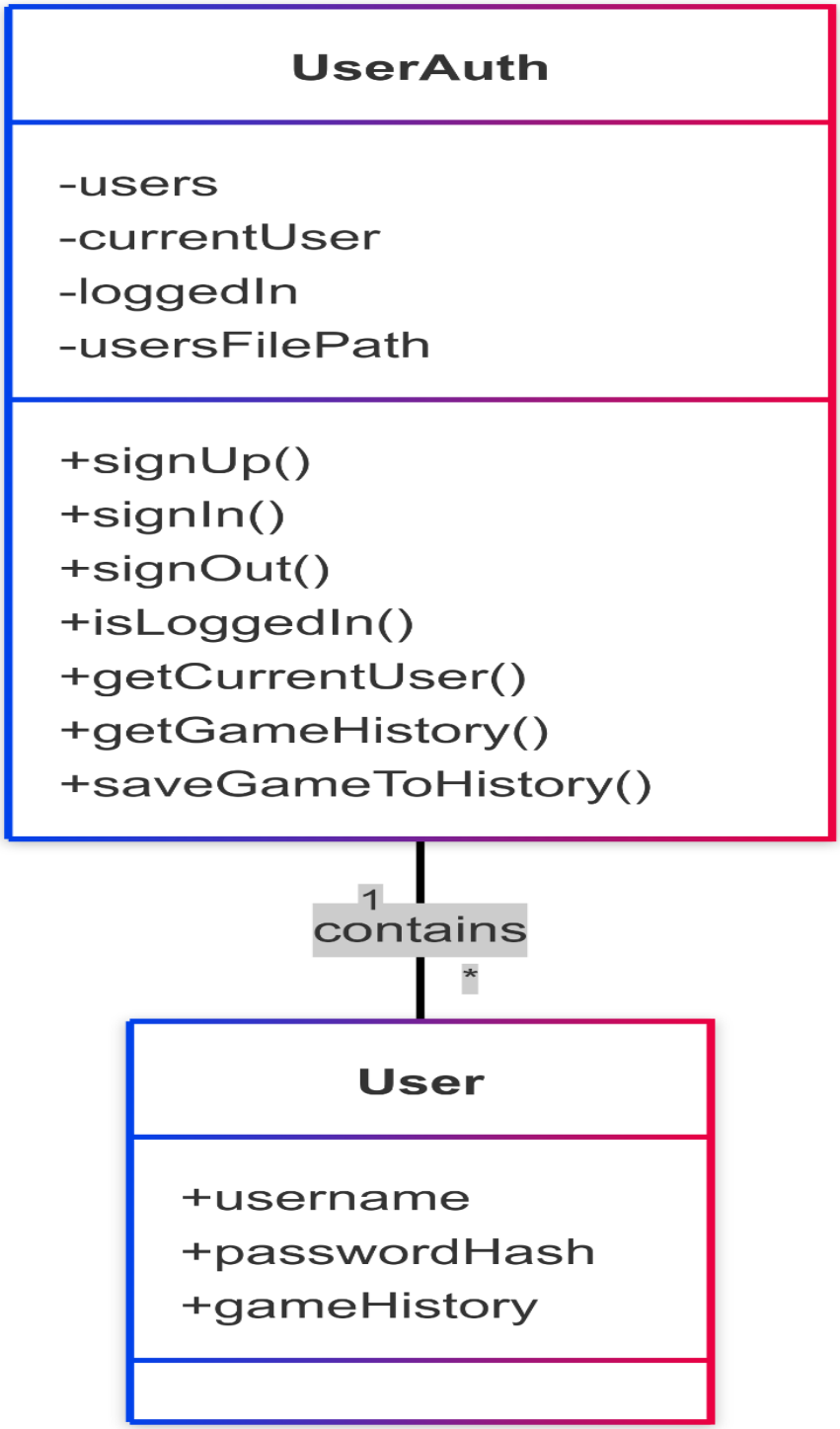
2.3.5 Data Storage

- **Responsibility:** Handles the persistent storage and retrieval of user data and game histories. Currently implemented using JSON files.
- **Key Technologies:** `QFile` , `QJsonDocument` , `QJsonObject` , `QJsonArray` .
- **Interfaces:** `loadUsersFromFile()` , `saveUsersToFile()` (within `UserAuth`).
- **Dependencies:** Used by `UserAuth` to manage user data.

3. Detailed Design

3.1 User Authentication Module (`UserAuth`)

3.1.1 Class Diagram



3.1.2 Class Descriptions

- **UserAuth :** **Attributes:**
  - ➢ users : A QMap storing User objects, keyed by username.
  - ➢ currentUser : The username of the currently logged-in user.
  - ➢ loggedIn : A boolean flag indicating the login status.
  - ➢ usersFilePath : The file path for storing user data (e.g., users.json).

  Methods:
  - ➢ UserAuth() : Constructor. Initializes file path and loads existing users.
  - ➢ ~UserAuth() : Destructor. Saves user data to file upon exit.
  - ➢ signUp(username, password) : Registers a new user. Hashes password and stores user data. Returns true on success, false if username exists or validation fails.
  - ➢ signIn(username, password) : Authenticates a user. Returns true on successful login, false otherwise.
  - ➢ isLoggedIn() : Checks if a user is currently logged in.
  - ➢ getCurrentUser() : Returns the username of the logged-in user.
  - ➢ signOut() : Logs out the current user.
  - ➢ saveGameToHistory(gameData) : Saves a completed game's data (as QJsonObject) to the current user's history.
  - ➢ getGameHistory() : Retrieves the game history for the current user.
  - ➢ hashPassword(password) : Private helper to hash passwords using
  - ➢ SHA-256.
  - ➢ loadUsersFromFile() : Private helper to load user data from users.json .
  - ➢ saveUsersToFile() : Private helper to save user data to users.json .
  - ➢ isValidPassword(password) : Private helper for password
  - ➢ complexity validation.
  - ➢ isValidUsername(username) : Private helper for username format validation.
- **User (struct/class):**

  Attributes:
  - ➢ username : User's unique identifier.
  - ➢ passwordHash : Hashed password.
  - ➢ gameHistory : QJsonArray storing a list of game records.

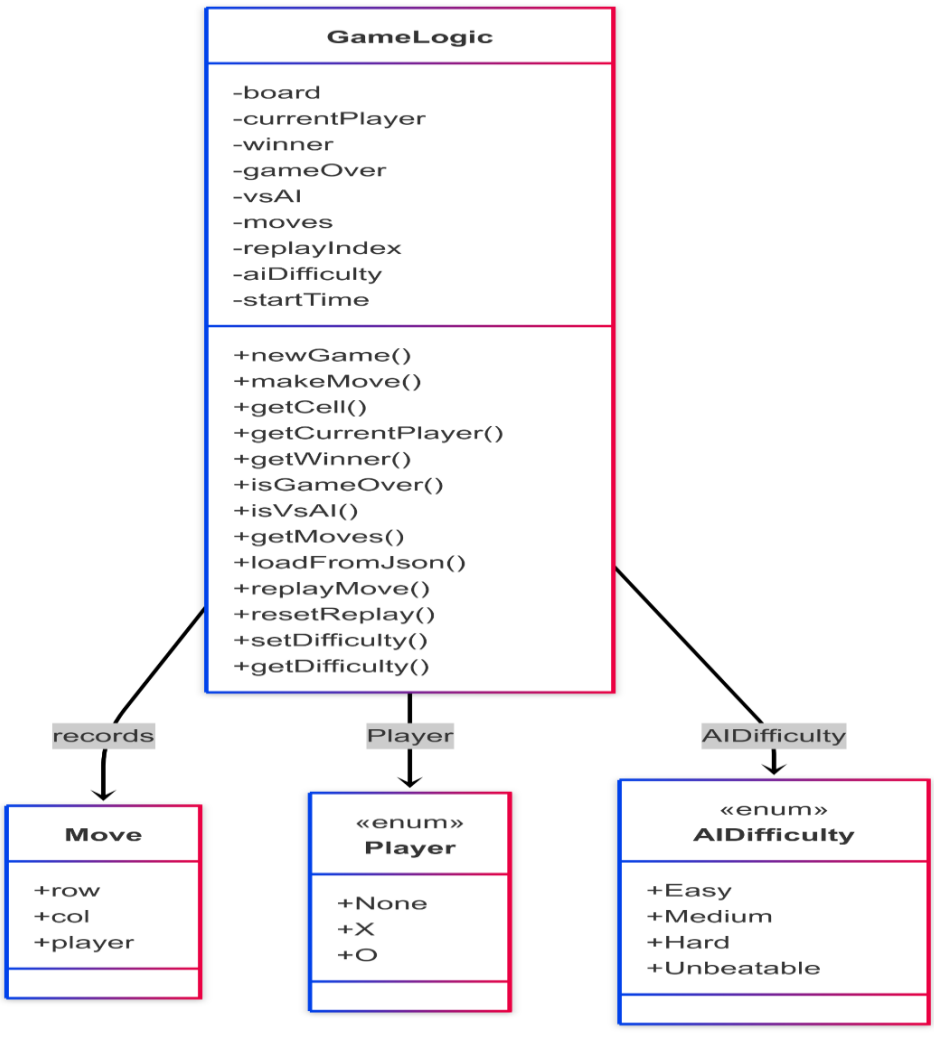
3.1.3 Data Storage Format (users.json)

User data is stored in a JSON file, where each top-level key is a username, and its value is a JSON object containing the `passwordHash` and `gameHistory` (an array of game objects).

```
{
  "user1": {
    "passwordHash": "hashed_password_of_user1",
    "gameHistory": [
      { "date": "...", "result": "...", "vsAI": true,
"difficulty": "Hard", "moves": [...] },
      { "date": "...", "result": "...", "vsAI": false, "moves": [...] }
    ]
  },
  "user2": {
    "passwordHash": "hashed_password_of_user2",
    "gameHistory": []
  }
}
```

3.2 Game Logic Module (`GameLogic`)

3.2.1 Class Diagram



3.2.2 Class Descriptions

- **GameLogic :** **Attributes:**
  - ➢ board : A 2D QVector representing the 3x3 Tic Tac Toe board.
  - ➢ currentPlayer : The player whose turn it is (Player::X or Player::O).
  - ➢ winner : The winning player (Player::X , Player::O , or Player::None for a tie/ongoing game).
  - ➢ gameOver : Boolean flag indicating if the game has ended.
  - ➢ vsAI : Boolean flag indicating if the game is agains AI.
  - ➢ moves : A QVector storing the sequence of Move objects for the current game.
  - ➢ replayIndex : Current index in the moves vector during game replay.
  - ➢ aiDifficulty : Current AI difficulty setting.
  - ➢ startTime : QDateTime when the game started, used for history.

Methods:

- ➢ GameLogic(parent) : Constructor. Initializes the board and sets currentPlayer to X .
- ➢ newGame(vsAI) : Resets the board and game state for a new game.
- ➢ makeMove(row, col) : Attempts to make a move at the specified coordinates. Validates the move, updates the board, records the move, checks for win/tie, switches player, and triggers AI move if applicable.
- ➢ setDifficulty(difficulty) : Sets the AI difficulty level. ▪
- ➢ getDifficulty() : Returns the current AI difficulty.
- ➢ getCell(row, col) : Returns the player at a specific cell. ▪
- ➢ getCurrentPlayer() : Returns the current player.
- ➢ getWinner() : Returns the winner of the game.
- ➢ isGameOver() : Checks if the game is over.
- ➢ isVsAI() : Checks if the current game is against AI.
- ➢ getGameAsJson() : Converts the current game state and history into a QJsonObject for saving.
- ➢ replayMove(index) : Replays the game up to a specific move index. ▪
- ➢ resetReplay() : Resets the replay to the beginning of the game.
- ➢ getMoves() : Returns the vector of recorded moves.
- ➢ loadFromJson(gameData) : Loads a game state from a QJsonObject for replaying.
- ➢ switchPlayer() : Private helper to switch the current player. ▪
- ➢ checkWin(row, col) : Private helper to check for a win condition after a move.
- ➢ checkGameOver() : Private helper to check for a tie condition.

- ➢ aiMove() : Private method to initiate an AI move.
- ➢ makeAIMove() : Private method that decides whether to use optimal or random AI move based on difficulty.
- ➢ shouldUseOptimalMove() : Private method to determine if AI should make an optimal move based on difficulty and randomness. ▪
- ➢ minimax(...) : Private recursive function implementing the Minimax algorithm with Alpha-Beta Pruning.
- ➢ isWin(board, player) : Private helper to check if a given player has won on a given board state. - getAvailableMoves(board) : Private helper to get all empty cells on a given board.
- ➢ evaluateBoard(board) : Private helper to evaluate a board state for the Minimax algorithm.

Signals:

- ➢ boardChanged() : Emitted when the board state changes, triggering GUI updates.
- ➢ gameEnded(winner) : Emitted when the game ends (win or tie).

Move (struct):

- ◦ **Attributes:**
  - ➢ row : Row of the move.
  - ➢ col : Column of the move.
  - ➢ player : Player who made the move.

Player (enum): Defines the possible states of a cell on the board: None , X , O .

AIDifficulty (enum): Defines the AI difficulty levels: Easy , Medium , Hard , Unbeatable .

3.3 GUI Module (MainWindow)

3.3.1 Class Diagram

3.3.2 Class Descriptions

❖ **Attributes:**

- **MainWindow :**
  - ➢ userAuth : Pointer to the UserAuth instance.
  - ➢ gameLogic : Pointer to the GameLogic instance.
  - ➢ stackedWidget : QStackedWidget to manage different views (Content truncated due to size limit. Use line ranges to read in chunks) instances for each page (login, signup, menu, game mode, game, history).
  - ➢ Pointers to various Qt GUI elements (QLineEdit , QPushButton , QLabel , QListWidget , QSlider , QComboBox) for user interaction and display.
  - ➢ boardButtons[3][3] : 2D array of QPushButton for the main game board.
  - ➢ boardReplayButtons[3][3] : 2D array of QPushButton for the game replay board.

❖ **Methods**

- ➢ MainWindow(auth, parent) : Constructor. Initializes UI, sets up pages, and connects signals/slots.
- ➢ ~MainWindow() : Destructor.
- ➢ setupLoginPage() , setupSignupPage() , setupMenuPage() , setupGameModePage() , setupGamePage() , setupHistoryPage() : Private methods to initialize and layout each UI page.
- ➢ handleLogin() , handleSignup() , handleLogout() : Slot methods to handle user authentication actions.
- ➢ handleCellClicked() : Slot method to handle clicks on game board cells.
- ➢ handleGameEnd(winner) : Slot method to handle game end event (win/tie).
- ➢ handleDifficultyChanged() : Slot method to handle AI difficulty
- ➢ selection.
- ➢ startTwoPlayerGame() , startAIGame() : Slot methods to start new games.
- ➢ loadGameHistory() : Loads and displays the current user's game history.
- ➢ loadSelectedGame() : Loads a selected game from history for replay.
- ➢ updateReplay(value) : Updates the replay board based on slider value.
- ➢ playNextMove() , playPreviousMove() : Navigate through game replay.
- ➢ updateBoard() : Updates the main game board display based on GameLogic state.
- ➢ updateBoardButtons() , updateReplayBoardButtons() : Private helpers to update the visual state of board buttons.
- ➢ showLoginPage() , showSignupPage() , showMenuPage() , showGamePage() , showGameModePage() , showHistoryPage() : Methods to switch between different UI pages.

3.4 AI Module (Integrated within `GameLogic`)

3.4.1 Minimax Algorithm with Alpha-Beta Pruning

The AI's decision-making is implemented within the `GameLogic` class, specifically in the `minimax` private method. This function recursively explores the game tree to find the optimal move for the AI (Player O), assuming the opponent (Player X) also plays optimally.

- **`minimax(board, depth, isMaximizing, alpha, beta)` :**
  - ➢ **`board`** : The current state of the Tic Tac Toe board.
  - ➢ **`depth`** : The current depth of the search tree. Used to penalize longer game paths.
  - ➢ **`isMaximizing`** : Boolean, `true` if the current call is for the maximizing player (AI, 'O'), `false` for the minimizing player (Human, 'X'). ◦
  - ➢ **`alpha`** : The best score that the maximizing player currently can guarantee at that level or above.
  - ➢ **`beta`** : The best score that the minimizing player currently can guarantee at
  - ➢ that level or above.
  - ➢ **Return Value**: The optimal score for the current board state.
- **BaseCases(Terminal States):**
  - ➢ If 'O' wins, return `10 - depth` (AI wins, prefer shorter paths).
  - ➢ If 'X' wins, return `depth - 10` (Human wins, AI wants to avoid this, prefer shorter paths to loss).
  - ➢ If it's a tie,return `0`.

- **RecursiveSteps:**

  - ➢ **Maximizing Player (AI):** Iterates through all empty cells, makes a hypothetical move for 'O', recursively calls `minimax` for the minimizing player, and takes the maximum score. Alpha-beta pruning is applied to cut off branches that won't affect the final decision.
  - ➢ **Minimizing Player (Human):** Similar to the maximizing player, but makes hypothetical moves for 'X', recursively calls `minimax` for the maximizing player, and takes the minimum score. Alpha-beta pruning is applied.

3.4.2 Difficulty Levels

The `shouldUseOptimalMove()` method in `GameLogic` introduces randomness based on the `aiDifficulty` setting. This allows the AI to make sub-optimal (random) moves at lower difficulty levels, providing a more human-like and less predictable opponent.

- ➢ **Easy:** 30% chance of optimal move.
- ➢ **Medium:** 50% chance of optimal move.
- ➢ **Hard:** 80% chance of optimalmove.
- ➢ **Unbeatable:** 100% chance of optimal move.

4. Data Structures and Algorithms

4.1 Core Data Structures

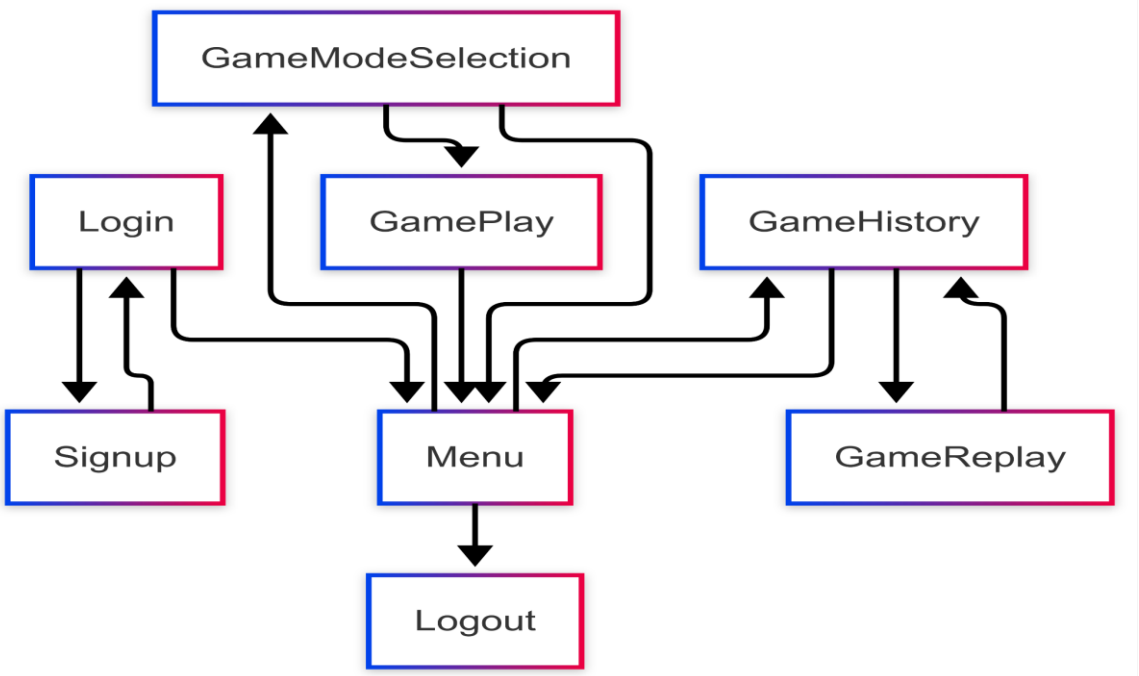
- **`QVector<QVector<Player>> board` :** Represents the 3x3 Tic Tac Toe board. Each inner `QVector` represents a row, and `Player` enum values (`None` , `X` , `O`) represent the cell content.
- **`QMap<QString, User> users` :** In `UserAuth` , stores user profiles. `QString` is the username (key), and `User` struct holds `passwordHash` and `gameHistory` .
- **`QJsonArray gameHistory` :** In `User` struct, stores a list of game records. Each record is a `QJsonObject` .
- **`QVector<Move> moves` :** In `GameLogic` , stores the sequence of moves made in a
- game, used for replay and saving history.

4.2 Key Algorithms

- **Password Hashing:** SHA-256 algorithm is used for secure storage of user passwords.
- **Minimax with Alpha-Beta Pruning:** The core AI algorithm for strategic move selection.
- **Game State Checking:** Algorithms for checking win conditions (rows, columns, diagonals) and tie conditions (full board).
- **JSON Serialization/Deserialization:** Used for saving and loading user data and game history to/from `users.json` .

5. User Interface Design

5.1 Page Flows



5.2 UI Elements and Layouts

- **Login Page:** Username and password input fields, Login button, Sign Up button, status label.
- **Sign Up Page:** Username, password, confirm password input fields, Sign Up button, Login button, status label.
- **Menu Page:** Welcome message, Play button, Game History button, Logout button.
- **Game Mode Selection Page:** "Player vs Player" button, "Player vs AI" button, AI Difficulty selection (combo box), Confirm Difficulty button, Back to Menu button.
- **Game Play Page:** Game board (3x3 grid of buttons), game status label (e.g., "Player X's Turn"), Back to Menu button.
- **Game History Page:** List of past games, "Replay Selected Game" button, Back to Menu button, replay controls (previous move, next move buttons, slider), replay status label, replay board.

5.3 Visual Design Considerations

- **Consistent Styling:** Use consistent fonts, colors, and button styles across all pages (as observed in `mainwindow.cpp` using Qt Style Sheets).
- **Clear Feedback:** Provide clear visual feedback for user actions (e.g., button clicks, cell marks on the board, status messages).
- **Responsive Layout:** While not explicitly stated as a requirement, a well-designed GUI should ideally adapt to different window sizes, though Qt's layout managers generally handle this well.

6. Testing Strategy

6.1 Unit Testing

Unit tests will be developed using the Qtest framework to ensure the correctness of individual components. Key areas for unit testing include:

UserAuth :

- `registerUser()` : Test successful registration, registration with existing
- username, and invalid inputs.
- `authenticateUser()` : Test successful authentication, incorrect password,
- and non-existent username.
- `getCurrentUser()` and `logout()` : Test user session management.

GameLogic :

- ➢ makeMove() : Test valid moves, invalid moves (already occupied cell, out of bounds), and turn switching.
- ➢ checkWin() : Test all possible win conditions (rows, columns diagonals) for both 'X' and 'O'.
- ➢ checkDraw() : Test draw conditions.
- ➢ resetGame() : Test board reset functionality.
- ➢ AI logic: Test AI's ability to make winning moves, block opponent's winning moves, and make optimal moves in various scenarios.

6.2 Integration Testing

Integration tests will focus on verifying the interactions between different modules:

- ➢ Login/Signup Flow: Test the complete flow from registration to login and menu navigation.
- ➢ Game Flow: Test starting a game, making moves, game ending (win/draw), and returning to the menu.
- ➢ History Management: Test saving and loading game history.

7. CI/CD Pipeline (GitHub Actions)

The project utilizes a comprehensive CI/CD pipeline configured with GitHub Actions, named "Advanced C++ CI/CD Pipeline with Qt 6". This pipeline automates the build, test, and deployment processes, ensuring code quality, reliability, and efficient delivery.

7.1 Workflow Overview

• The workflow is triggered automatically on every `push` and `pull_request` event to the `main` and `master` branches and consists of two main jobs: build and deploy ..

7.1.1 Build Job

This job is responsible for compiling the application, running static analysis, and executing tests. It runs on an `ubuntu-22.04` environment and includes the following steps:

- ➢ Checkout code: Retrieves the latest code from the repository.
- ➢ Install Qt 6 and dependencies: Installs necessary Qt 6 development packages,

qmake6 , cmake , g++ , build-essential , and testing/analysis tools like cppcheck and valgrind . This ensures a consistent build environment.

- ➢ Verify Qt 6 installation: Confirms that qmake6 is correctly installed and accessible.
- ➢ Code Quality Analysis: Executes cppcheck for static code analysis to identify potential bugs, code smells, and security vulnerabilities. The results are output to cppcheck_report.xml .
- ➢ Security Scan: Performs a basic security scan by searching for common sensitive keywords (e.g., "password", "secret", "key") within the source code. This is a preliminary check for accidental credential exposure.
- ➢ Build project with qmake6: Compiles the Tic Tac Toe application using qmake6 and make .

- ➢ Run Comprehensive Tests:
  - ❖ Unit Tests: Builds and runs unit tests located in the ./Unittests_tests directory. Test results are saved in JUnit XML format to test_results/unit_tests.xml .
  - ❖ Integration Tests: Builds and runs integration tests located in the ./ Integration_Test directory. Test results are saved in JUnit XML format to test_results/integration_tests.xml .
- ➢ Performance Benchmarking: Executes the compiled application and uses / usr/bin/time -v to measure performance metrics such as maximum resident set size, user time, and system time. A summary is saved to performance_summary.txt .
- ➢ Prepare Deployment Package: Creates a deployment_package directory and copies the compiled executable (Embedded) and any configuration files (.conf , .ini , .db) into it. A build_info.txt file containing build date, commit hash, and branch information is also generated.
- ➢ Upload Comprehensive Artifacts: Uploads all generated artifacts, including the executable, deployment package, test results, performance reports, • 1. 2. 3. 4. 5. 6. 7. ▪ ▪ 8. 9. 10. cppcheck report, and build files, as a GitHub Action artifact. These artifacts are retained for 30 days.

7.1.2 Deploy Job

This job is responsible for deploying the application. It depends on the successful completion of the build job and is triggered only on pushes to the main branch. It also runs on an ubuntu-22.04 environment

Steps:

- ➢ Checkout code: Retrieves the latest code from the repository.
- ➢ Download build artifacts: Downloads the artifacts generated by the build job.
- ➢ Prepare Deployment Directory: Organizes the downloaded artifacts into a deployment directory, ensuring the executable has appropriate permissions. A DEPLOYMENT_INFO.txt file with deployment details is created.
- ➢ Create Release Archive (.zip): Compresses the contents of the deployment directory into a .zip archive for easy distribution.
- ➢ Package Test Results: Compresses the test results into a separate .zip archive.
- ➢ Create GitHub Release Tag: Configures Git user information and creates an annotated Git tag for the release (e.g., v1 , v2 , etc., based on the GitHub run number). This tag is then pushed to the remote repository.
- ➢ Deploy Application: Copies the prepared deployment package to a temporary production directory (/tmp/tic-tac-toe-production) on the runner. Note: In a real-world scenario, this step would involve deploying to a production server or distribution platform.
- ➢ Verify Deployment: Checks for the presence of the deployed executable to confirm successful deployment.
- ➢ Upload Release Artifacts: Uploads the generated release .zip package, the deployment directory, and the test results package as GitHub Action artifacts.
- ➢ Upload to GitHub Release: Creates a new GitHub Release, attaching the release .zip package, the contents of the deployment directory, and the test results package. The release body includes a brief description and references the build number

8. Future Considerations

8.1 Enhancements

- **Online Multiplayer:** Implement network communication for player-vs-player games over the internet.
- **Advanced AI:** Explore more sophisticated AI algorithms or machine learning approaches for the AI opponent.
- **Customizable Board Sizes:** Allow users to select different board dimensions (e.g., 4x4, 5x5).
- **User Profiles and Statistics:** Expand user profiles to include game statistics (wins, losses, ties, win rates).

8.2 Performance Optimization

- Further optimize the Minimax algorithm for larger board sizes or deeper search depths if performance becomes an issue.
- Profile application performance to identify and address bottlenecks.

8.3 Security Enhancements

- Consider more robust password management techniques (e.g., salting).
- Implement input validation at all layers to prevent potential vulnerabilities.

9. Conclusion

This SDS provides a comprehensive design for the Advanced Tic Tac Toe game, covering its architecture, component interactions, and detailed module designs. By adhering to this specification, the development team can ensure a structured and efficient implementation process, leading to a robust and maintainable software product. The design emphasizes modularity, testability, and adherence to best practices, laying a solid foundation for future enhancements and scalability.