Tic Tac Toe Game: Integration Testing Documentation

1. Introduction

This document provides comprehensive testing documentation for the advanced Tic Tac Toe game, focusing specifically on the integration testing efforts. The game, developed in C++, incorporates features such as user authentication, personalized game history, and an intelligent AI opponent. As part of a commitment to robust software engineering practices, rigorous testing, including both unit and integration testing, is a core component of the development lifecycle. This documentation serves to detail the integration test cases, their objectives, and expected outcomes, ensuring the seamless interaction and functionality of various system components.

2. Scope

The scope of this document is limited to the integration testing of the Tic Tac Toe game. It outlines the methodology, environment, and specific test cases executed to validate the interactions between different modules, such as user authentication, game logic, and the graphical user interface (GUI). Unit testing and CI/CD pipeline configurations, while crucial to the project, are addressed in separate documentation.

3. Testing Strategy

The integration testing strategy adopted for the Tic Tac Toe game is a top-down approach, focusing on verifying the interactions between integrated modules. This approach is particularly effective for systems where the overall architecture and module interfaces are well-defined. By starting with higher-level components (e.g., GUI interacting with game logic and user authentication) and progressively testing their integration with lower-level modules, we can identify interface defects and ensure that the system functions as a cohesive unit. The primary tool used for integration testing in this project is Qt test, a comprehensive C++ test framework.

4. Detailed Integration Test Cases

This section provides a detailed breakdown of each integration test case, including its purpose, the steps taken to execute the test, and the expected outcomes. These tests are designed to simulate real-world user interactions and system behaviors, ensuring the seamless operation of the integrated components.

4.1. Test Case: `testLoginAndStartAIGame`

- **Purpose:** This test case is designed to thoroughly verify the user authentication process, specifically focusing on user login and the subsequent initiation of a player-versus-AI game. It ensures that new users can be registered, existing users can log in successfully, and the game system correctly transitions into an AI game mode.

- **Preconditions:** A clean testing environment where user authentication data can be managed without interference from previous test runs.
- **Steps:**
    1. **User Authentication Setup:** An instance of `UserAuth` is created to manage user accounts. A predefined `username` ("testuser") and `password` ("Password1!") are used for consistency.
    2. **User Existence Check and Registration/Login:** The test first attempts to sign in the `testuser`. If the sign-in fails (indicating the user does not exist), the user is registered using `auth.signUp()`, followed by another attempt to sign in. This ensures that the test can run independently, creating the user if necessary.
    3. **Authentication Verification:** Assertions (`QVERIFY(auth.isLoggedIn())` and `QCOMPARE(auth.getCurrentUser(), username)`) are used to confirm that the user is indeed logged in and that the `getCurrentUser()` method returns the correct username.
    4. **GUI Initialization:** A `MainWindow` object is instantiated, simulating the main application window. This window is associated with the `UserAuth` instance to ensure proper user context.
    5. **Navigation to Game Mode Selection:** The `window.showMenuPage()` and `window.showGameModePage()` methods are called to simulate navigation through the application's user interface to reach the game mode selection screen.
    6. **AI Difficulty Setting:** The AI difficulty is explicitly set to `AIDifficulty::Easy` using `window.gameLogic->setDifficulty()`. This ensures a predictable AI behavior for testing purposes.
    7. **AI Game Start:** The `window.startAIGame()` method is invoked to initiate a game session against the AI opponent.
- **Expected Results:**
    - **Successful Login:** The `auth.isLoggedIn()` method must return `true`, confirming that the user has been successfully authenticated and a session has been established.
    - **Correct User Identification:** The `auth.getCurrentUser()` method must return the exact `username` ("testuser"), verifying that the correct user profile is active.
    - **AI Game Mode Activation:** The `window.gameLogic->isVsAI()` method must return `true`, indicating that the game has correctly transitioned into the player-versus-AI mode.
    - **Game Not Over:** The `window.gameLogic->isGameOver()` method must return `false` immediately after starting the game, confirming that the game is in an active state and awaiting player input.

4.2. Test Case: `testGameLogicVsAI`

- **Purpose:** This test case is designed to thoroughly evaluate the core game logic when a human player interacts with an AI opponent. It specifically focuses on verifying the `makeMove` functionality for both human and AI players, and ensuring that the game board accurately reflects these moves.
- **Preconditions:** A properly initialized `GameLogic` instance, capable of simulating game states and AI responses.
- **Steps:**
    1. **Game Logic Initialization:** An instance of `GameLogic` is created. The AI difficulty is set to `AIDifficulty::Medium` to ensure a more challenging, yet predictable, AI behavior.
    2. **New Game Start:** A new game is initiated against the AI by calling `logic.newGame(true)`, where `true` signifies a player-versus-AI match.
    3. **Human Player Move:** The `logic.makeMove(0, 0)` method is called to simulate a human player placing their mark ('X') at the top-left corner of the Tic Tac Toe board. The return value is verified to ensure the move was valid and successful.
    4. **AI Move Verification:** After the human player's move, the AI is expected to make its move automatically. The test then checks the size of the `logic.getMoves()` list, asserting that it contains at least two moves (one by the human, one by the AI). This implicitly verifies that the AI has responded.
    5. **Board State Verification (Human Move):** The `logic.getCell(0, 0)` method is used to retrieve the player mark at the (0,0) position. It is asserted that this cell contains `Player::X`, confirming the human player's move.
    6. **Board State Verification (AI Move):** A nested loop iterates through all cells of the 3x3 board. It checks if any cell other than (0,0) contains `Player::O` (the AI's mark). A boolean flag `aiMoved` is set to `true` if an AI move is found. Finally, `QVERIFY(aiMoved)` asserts that the AI has indeed placed its mark on the board.
- **Expected Results:**
    - **Successful Human Move:** The `logic.makeMove(0, 0)` call must return `true`, indicating that the human player's move was successfully registered.
    - **AI Response:** The `logic.getMoves().size()` must be greater than or equal to 2, confirming that the AI has made a move in response to the human player.
    - **Correct Board State (Human):** The cell at (0,0) must contain `Player::X`, verifying the accurate placement of the human player's mark.
    - **Correct Board State (AI):** At least one cell on the board (excluding (0,0)) must contain `Player::O`, confirming that the AI has made a valid move and updated the board state accordingly.

4.3. Test Case: `testGameEndAndHistory`

- **Purpose:** This test case is crucial for validating the end-game conditions and the persistence of game data. It ensures that the game correctly identifies when it has

concluded (either by a win or a tie), and that the game's outcome and moves are accurately saved to and retrieved from the user's personalized game history.

- **Preconditions:** A functional `UserAuth` system for managing user accounts and game history, and a `GameLogic` instance capable of simulating a complete game.
- **Steps:**
    1. **User Authentication Setup:** An instance of `UserAuth` is created. A distinct `username` ("testuser2") and `password` ("Password2!") are used to avoid conflicts with other tests.
    2. **User Existence and Login:** Similar to `testLoginAndStartAIGame`, the test ensures that `testuser2` is logged in, registering the user if they do not already exist.
    3. **Game Initialization:** A `GameLogic` instance is created, and the AI difficulty is set to `AIDifficulty::Easy`. A new game is started against the AI.
    4. **Simulate Full Game Play:** A `while` loop continues as long as `logic.isGameOver()` returns `false`. Inside the loop, the test iterates through the 3x3 board to find the first empty cell (`Player::None`). A move is then simulated at this empty cell using `logic.makeMove()`. This process continues until the game reaches an end state.
    5. **Game Over Verification:** After the loop, `QVERIFY(logic.isGameOver())` asserts that the game has indeed concluded, either by a win for one player or a tie.
    6. **Save Game to History:** The completed game's data is retrieved as a `QJsonObject` using `logic.getGameAsJson()`. This JSON object is then saved to the logged-in user's history using `auth.saveGameToHistory()`. The success of this operation is verified.
    7. **Retrieve and Verify History:** The entire game history for the current user is retrieved as a `QJsonArray` using `auth.getGameHistory()`. The test then performs several assertions:
        - `QVERIFY(history.size() > 0)`: Ensures that the history is not empty.
        - `QJsonObject lastGame = history.last().toObject()`: Retrieves the most recently saved game from the history.
        - `QVERIFY(lastGame.contains("result"))`: Verifies that the saved game data contains a "result" field.
        - `QVERIFY(lastGame.contains("moves"))`: Verifies that the saved game data contains a "moves" field.
        - `QVERIFY(lastGame["moves"].toArray().size() > 0)`: Ensures that the "moves" array within the saved game data is not empty, confirming that the game's moves were recorded.
- **Expected Results:**
    - **Game Conclusion:** The game must reach an `isGameOver()` state, indicating a definitive end to the game session.

- **Successful History Save:** The `auth.saveGameToHistory()` method must return `true`, confirming that the game data was successfully stored.
- **History Retrieval and Content:** The `auth.getGameHistory()` must return a non-empty array. The last game in this history must contain valid "result" and "moves" fields, and the "moves" array must not be empty, thereby validating the integrity and completeness of the saved game data.

5. Conclusion

This integration testing documentation highlights the robust testing efforts undertaken for the advanced Tic Tac Toe game. The detailed test cases, as implemented in `integration_tests.cpp`, demonstrate a commitment to ensuring the seamless interaction of critical system components, including user authentication, game logic, and AI integration. By systematically verifying these interactions, we can confidently assert the stability and reliability of the game. The use of QtTest has provided a solid framework for these tests, contributing significantly to the overall quality assurance process. Continued adherence to these testing practices will be essential for future development and maintenance of the game.