Testing Documentation

for Tic-Tac-Toe

Table of Contents

Definitions, Acronyms, and Abbreviations

AI: Artificial Intelligence

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).

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.

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.

TDD: Test-Driven Development. A software development process relying on software requirements being converted to test cases before software is fully developed.

QTest: The unit testing framework provided by Qt.

UT: Unit Test / Unit Testing.

IT: Integration Test / Integration Testing.

SUT: System Under Test.

1. Introduction

1.1. Purpose

This Testing Documentation details the strategies, methodologies, and results of the testing efforts for the Tic-Tac-Toe application. Its purpose is to ensure the quality, reliability, and correctness of the software by verifying that it meets the specified functional and non-functional requirements. This document covers both unit and integration testing approaches, leveraging the Qt Test framework.

1.2. Scope

The scope of this document encompasses the testing of all core logic components and the verification of key interactions within the Tic-Tac-Toe application. This includes testing of game mechanics, AI behavior, user management functionalities, and game history persistence. The User Interface (UI) is primarily verified through integration testing with its underlying logic.

2. Testing Strategy

2.1. Overall Approach

The testing strategy adopts a multi-level approach, primarily focusing on unit testing for individual components and integration testing for verifying interactions between these components. This strategy aims for early detection of defects, ensuring robustness at foundational levels before integrating into a larger system. The testing process utilizes the Qt Test framework, enabling automated and repeatable tests. A strong emphasis is placed on covering the critical paths and edge cases for each tested module. The project structure, with a dedicated `tests.pro` file, facilitates a clean separation of test code from application code.

3. Unit Testing

Unit testing focuses on verifying the smallest testable parts of the application in isolation. Each core class has a dedicated test suite using Qt Test. The test runner `main.cpp` executes all unit test suites.

3.1. Test Framework

The application's unit tests are implemented using **Qt Test**, a lightweight and powerful unit testing framework provided by the Qt library. Qt Test provides macros like `QCOMPARE` for comparing values and `QVERIFY` for asserting conditions, enabling clear and concise test case definitions. The test environment is configured to run as a console application, reporting results directly to the command line.

3.2. Test Cases

Detailed test cases have been developed for each core logical class, ensuring comprehensive coverage of their functionalities.

3.2.1. Board Class Tests

The `TestBoard` class is responsible for unit testing the `Board` class, which manages the Tic-Tac-Toe game state.

- **Initial State Verification:** Tests `testInitialBoardState()` to confirm that a newly created board is completely empty.
- **Move Validation:** `testMakeMoveValid()` verifies successful placement of marks on empty cells, while `testMakeMoveInvalid()` checks that moves on occupied cells or out-of-bounds positions are correctly rejected.
- **Win Condition Checks:** Comprehensive tests are implemented for `testCheckWinRows()`, `testCheckWinColumns()`, and `testCheckWinDiagonals()` to ensure accurate detection of winning lines for both Player X and Player O across all possible alignments.
- **No Win Condition:** `testCheckWinNoWin()` verifies that no winner is declared when the board state does not contain a winning line.
- **Full Board Detection:** `testIsFull()` checks if the board correctly reports itself as full when all cells are occupied, and not full otherwise.
- **Board Reset:** `testResetBoard()` confirms that the `reset()` method correctly clears the board to its initial empty state.

3.2.2. AIPlayer Class Tests

The `TestAIPlayer` class focuses on validating the decision-making logic of the `AIPlayer`.

- **Winning Move Priority:** `testAiShouldWinWhenPossible()` ensures that the AI correctly identifies and chooses an immediate winning move when one is available. This confirms the AI's primary objective.
- **Blocking Move Priority:** `testAiShouldBlockPlayerWin()` verifies that the AI prioritizes blocking an opponent's immediate winning move over other options. This confirms the AI's defensive strategy.

3.2.3. GameLogic Class Tests

The `TestGameLogic` class performs unit tests on the `GameLogic` class, which orchestrates the game flow and rules.

- **Game Initialization:** `testStartGame()` checks that the game starts correctly, setting the initial player (Player X) and the game mode (PvP or Vs AI).

- **Player Move Handling:** `testHandlePlayerMove()` validates the core move processing, including valid and invalid moves, player switching, and the transition to game end states (win/draw).
- **Game Reset Functionality:** `testResetGame()` verifies that the game state, current player, and move history are correctly cleared when a game reset is initiated.
- **Current Player Tracking:** `testGetCurrentPlayer()` ensures that the `currentPlayer` is correctly tracked and switched after valid moves.
- **Winner Determination:** `testGetWinner()` verifies that the game logic correctly identifies a winner (Player X or O) or a draw state after a series of moves.
- **AI Mode Check:** `testIsVsAI()` confirms that the game logic correctly identifies whether the game is configured in Player vs. AI mode.

3.2.4. DatabaseManager Class Tests

The `TestDatabaseManager` class is dedicated to testing the `DatabaseManager` class, which handles all database interactions. Robust setup and teardown methods (`initTestCase`, `cleanupTestCase`, `init`, `cleanup`) ensure that each test runs on a clean and isolated database instance ("test_users.db").

- **Database Initialization:** `testInitializeDatabase()` verifies that the database connection is established and that the `users` and `game_history` tables are created correctly.
- **User Registration:** `testRegisterUser_success()` confirms successful user registration, while `testRegisterUser_duplicate()` ensures that attempts to register with an existing username fail.
- **User Authentication:** `testAuthenticateUser_success()` validates successful login with correct credentials, and `testAuthenticateUser_failure()` checks for correct rejection with invalid passwords or non-existent usernames.
- **Password Reset:** `testResetUserPassword_success()` verifies that a user's password can be successfully changed and that the new password works, while `testResetUserPassword_nonExistentUser()` ensures attempts for non-existent users are rejected.
- **User Information Retrieval:** `testGetUserInfo()` confirms that user details (first name, last name, username) can be correctly fetched.
- **Game History Management:** `testSaveAndLoadGameHistory()` validates that game outcomes and move sequences are correctly saved and retrieved, including proper ordering. `testLoadGameHistory_noHistory()` checks behavior for users with no saved games, and `testDeleteGameHistory()` verifies the removal of specific game records.
- **Move String Retrieval:** `testGetGameMoves()` ensures that the full move string for a specific game ID can be accurately retrieved.

4. Integration Testing

Integration testing focuses on verifying the interactions and data flow between different modules and components of the application.

4.1. Integration Approach

A top-down integration approach is employed, where higher-level modules are tested with their immediate subordinates. The `MainWindow` serves as the integration point for much of the system, connecting user interface actions to the `GameLogic` and `DatabaseManager` components via Qt's signal-slot mechanism. This approach ensures that the primary user workflows are functional.

4.2. Key Integration Scenarios

- **User Authentication Flow:**
  - **Scenario:** Successful user registration followed by successful login and subsequent navigation to the main menu.
  - **Verification:** User account is created in the database, login redirects correctly, and welcome messages are displayed.
- **Full Game Play (PvP):**
  - **Scenario:** Two human players complete a full game from start to finish (win or draw).
  - **Verification:** Moves are registered, players switch turns, winning/draw conditions are detected, and the game outcome is correctly displayed.
- **Full Game Play (PvAI - Hard):**
  - **Scenario:** A human player plays against the "Hard" AI from start to finish.
  - **Verification:** AI makes moves, game progresses to a win or draw, and AI's moves are optimal (observable from play). The AI delay is also observed.
- **Game History Persistence and Replay:**
  - **Scenario:** Play a game, save its history, log out, log back in, load history, and replay the saved game.
  - **Verification:** Game record appears in history, replay correctly animates moves on the board, and board interaction is disabled during replay.
- **Password Reset Functionality:**
  - **Scenario:** User requests a password reset, provides new credentials, and then successfully logs in with the new password.
  - **Verification:** Password update is successful in the database, and old password no longer authenticates.

5. Test Coverage

5.1. Code Coverage

Code coverage aims to measure the extent to which the application's source code is executed by the test suite. While specific coverage reports (e.g., statement, branch coverage) are not part of this document, the unit test suite has been designed with the intent of achieving high coverage for the core logic components (`Board`, `AIPlayer`, `GameLogic`, `DatabaseManager`). The organization of tests, with dedicated test classes for each module, supports a systematic approach to identifying and filling coverage gaps. The `tests.pro` file specifically compiles the application's core source files alongside the test files, allowing for comprehensive instrumentation and analysis if a coverage tool (e.g., GCOV, LCOV) is integrated into the build process.

6. Test Results Summary

6.1. Current Test Status

As of this documentation, all implemented unit tests within the `TestBoard`, `TestAIPlayer`, `TestGameLogic`, and `TestDatabaseManager` suites are expected to pass consistently, indicating that individual components function correctly in isolation. Integration tests demonstrate that the major components interact as designed, supporting the primary user workflows such as user authentication, game play in both modes, and game history management. No critical defects or regressions are currently known based on the existing test suite executions.