Testing Documentation
Tic Tac Toe Game

Supervised by: Dr. Omar Nasr
Date: June 2025

1. Introduction

The testing documentation for the Advanced Tic Tac Toe Game offers an in-depth review of the testing efforts aimed at ensuring the game's functionality, reliability, and performance. This section details the purpose of the testing documentation, the target audience, and recommendations for understanding the testing process more effectively.

1.1 Purpose

The objectives of this testing documentation are to:

- - Present a comprehensive account of the testing strategies, test cases, and coverage reports conducted for the Advanced Tic Tac Toe Game.
- - Confirm the accuracy, dependability, and performance of the game's features through thorough testing.
- - Serve as a reference for developers, testers, and project stakeholders to evaluate the game's quality and readiness for release.

2. Testing Strategies

2.1 Unit Testing

Unit testing involves testing individual units or components of the software in isolation to ensure their correctness and functionality. The focus is on validating key components such as the Game class, Player class, AI algorithms, and other essential parts. All tests are conducted within a single class called Test.

Approach

- The Test class covers:
- Game Logic Tests:
    - Board Initialization: Verifies that the game board is correctly set up at the start.
    - Player Moves: Ensures that player moves are accurately processed and reflected on the board.
    - Win/Tie Detection: Confirms the game correctly identifies win or tie conditions.
    - Game Outcomes: Validates that the game reports the correct outcome after each game.
- Player Action Tests:
    - Move Validation: Checks that player moves are validated correctly.
    - Profile Management: Ensures player profiles are created and updated properly.
    - Interface Interaction: Verifies proper interaction with the game interface.
- AI Behavior Tests:
    - Algorithm Implementation: Assesses AI behavior using the minimax algorithm with alpha-beta pruning.

    - Difficulty Levels: Ensures the AI functions correctly across different difficulty levels.

- Session Management Tests:

    - Game Addition: Confirms new games are added correctly.
    - Score Updates: Ensures scores update accurately.
    - Data Retrieval: Validates correct retrieval of session data.

- Database Operation Tests:

    - Data Saving: Ensures game data and profiles are saved correctly.
    - Data Retrieval: Checks accurate retrieval from the database.

- Game History Tests:

    - Record Accuracy: Confirms correct recording of past games.
    - Storage Integrity: Ensures data is stored without corruption.
    - Review Access: Confirms users can access and review history.

Tool

Qt Test Framework is used to write and run these unit tests, providing structured and reliable validation of individual components.

2.2 Integration Testing

Integration testing ensures different modules interact and integrate correctly. It focuses on seamless component communication and data flow—helping to uncover issues not visible in unit tests.

Tool

Integration tests are performed using Qt Test Framework and GitHub Actions (.yaml files).

3. Coverage Report

Coverage reports provide insights into which parts of the code were tested and highlight untested areas.

Purpose

- - Measure the percentage of code executed during tests.
- - Identify untested sections.
- - Guide additional test writing to improve coverage.

Tools

- - Qt Test Framework: For executing tests.
- - Coverage Tools: Tools like gcov, lcov, or Qt Creator’s built-in coverage tools.

3.1 Key Coverage Metrics

- Statement Coverage: Measures the percentage of executed statements.
- Branch Coverage: Measures how many decision branches were tested.
- Function Coverage: Measures the percentage of functions invoked during tests.
- Path Coverage: Measures the percentage of execution paths tested.

Coverage Analysis Process

1. Run Tests: using Qt Test framework.
2. Generate Reports: using coverage tools.
3. Analyze Data: to detect gaps in coverage.
4. Identify Gaps: in untested code.
5. Improve Coverage: by adding or enhancing test cases, then re-running and re-analyzing.

4. Conclusion

The testing strategy effectively identified and resolved potential issues, resulting in a highly reliable and functional game. By applying comprehensive unit and integration testing along with detailed coverage analysis, all critical aspects of the game were thoroughly validated.