Tic-Tac-Toe QTest Unit Testing Documentation

Introduction

This documentation provides a comprehensive analysis of the unit testing suite for a Tic-Tac-Toe game application built using Qt framework. The testing suite consists of main test file:

- **test_ai.cpp**: Focused on testing the AI game logic and algorithms

---

Testing Framework Overview

QTest Framework Features Used

The tests leverage several key QTest features:

1. **Test Class Structure**: Both test classes inherit from QObject and use the Q_OBJECT macro
2. **Test Fixtures**: Setup and cleanup methods (initTestCase, cleanupTestCase, init, cleanup)
3. **Assertions**: Various QVERIFY and QCOMPARE macros for validating test conditions
4. **Test Organization**: Logical grouping of test methods by functionality
5. **Automatic Test Discovery**: Uses QTEST_MAIN and QTEST_APPLESS_MAIN macros

Test Execution Flow

initTestCase() → [init() → testMethod() → cleanup()] × N → cleanupTestCase()

AI Logic Tests (test_ai.cpp)

Overview

The TestGameLogic class focuses on testing the core AI algorithms and game logic for the Tic-Tac-Toe game. It tests the ai class which appears to implement the game's artificial intelligence.

Test Categories

1. Basic Game State Tests

testEmptyBoard()

```
void testEmptyBoard() {
    ai testAI(nullptr);
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j)
            testAI.board[i][j] = ' ';
    QVERIFY(!testAI.checkWin("X"));
    QVERIFY(!testAI.checkWin("O"));
    QVERIFY(!testAI.isBoardFull());
}
```

Purpose: Validates that an empty board correctly reports no winners and is not full.

Key Assertions:

- Neither player should be winning on an empty board
- Empty board should not be reported as full

2. Win Condition Tests

testHorizontalWinConditions()

```
// Test X wins in first row
testAI.board[0][0] = 'X'; testAI.board[0][1] = 'X'; testAI.board[0][2] = 'X';
testAI.board[1][0] = 'O'; testAI.board[1][1] = ' '; testAI.board[1][2] = 'O';
testAI.board[2][0] = ' '; testAI.board[2][1] = ' '; testAI.board[2][2] = ' ';
QVERIFY(testAI.checkWin("X"));
QVERIFY(!testAI.checkWin("O"));
```

Purpose: Tests all three horizontal win patterns (rows 0, 1, 2) for both X and O players.

Coverage:

- Row 0: X-X-X win condition
- Row 1: O-O-O win condition

- Row 2: X-X-X win condition

testVerticalWinConditions()

// Test X wins in first column

testAI.board[0][0] = 'X'; testAI.board[0][1] = 'O'; testAI.board[0][2] = ' ';

testAI.board[1][0] = 'X'; testAI.board[1][1] = ' '; testAI.board[1][2] = 'O';

testAI.board[2][0] = 'X'; testAI.board[2][1] = 'O'; testAI.board[2][2] = ' ';

Purpose: Tests all three vertical win patterns (columns 0, 1, 2) for both players.

Coverage:

- Column 0: X-X-X win condition
- Column 1: O-O-O win condition
- Column 2: X-X-X win condition

testDiagonalWinConditions()

// Test X wins main diagonal (top-left to bottom-right)

testAI.board[0][0] = 'X'; testAI.board[0][1] = 'O'; testAI.board[0][2] = ' ';

testAI.board[1][0] = ' '; testAI.board[1][1] = 'X'; testAI.board[1][2] = 'O';

testAI.board[2][0] = 'O'; testAI.board[2][1] = ' '; testAI.board[2][2] = 'X';

Purpose: Tests both diagonal win conditions.

Coverage:

- Main diagonal (top-left to bottom-right): X-X-X
- Anti-diagonal (top-right to bottom-left): O-O-O

3. Game State Analysis Tests

testTieGame()

```
char draw[3][3] = {
    {'X', 'O', 'X'},
    {'O', 'O', 'X'},
    {'X', 'X', 'O'}
};
```

Purpose: Tests a complete game that ends in a draw.

Validation:

- No player should be winning
- Board should be reported as full
- Represents a realistic tie scenario

testBoardFullDetection()

Purpose: Tests the board full detection algorithm.

Test Cases:

- Partially filled board (should return false)
- Completely filled board (should return true)

4. AI Algorithm Tests

testMinimaxEvaluation()

```
// Test AI winning position evaluation
char aiWinBoard[3][3] = {
    {'O', 'O', 'O'},
    {'X', 'X', ' '},
    {' ', ' ', ' '}
};
int score = testAI.evaluate(aiWinBoard);
QVERIFY(score > 0); // AI should have positive score when winning
```

Purpose: Tests the minimax evaluation function used by the AI.

Test Scenarios:

- AI winning position (should return positive score)
- Player winning position (should return negative score)
- Neutral position (should return zero score)

Algorithm Validation: Ensures the AI correctly evaluates board positions for decision-making.

5. Edge Case and Error Handling Tests

testEdgeCases()

Purpose: Tests boundary conditions and unusual scenarios.

Coverage:

- Single move scenarios
- Almost full board conditions
- Early game states

testGameStateConsistency()

Purpose: Validates logical consistency of game states.

Key Validations:

- A winning board cannot simultaneously be a draw
- Both players cannot win at the same time
- Game state invariants are maintained

testInvalidGameStates()

Purpose: Tests how the system handles impossible game states.

Scenario: Both players appear to have winning conditions simultaneously.

Note: While this shouldn't occur in normal gameplay, testing edge cases ensures robustness.