# 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 two main test files:

- **test_ai.cpp**: Focused on testing the AI game logic and algorithms
- **test_components.cpp**: Comprehensive testing of game components, data structures, and user management

Both test files utilize Qt's QTest framework, which provides a robust testing environment for Qt applications with features like test case organization, assertions, and automatic test execution.

---

### 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.

---

**Integrated Component Tests (test_components.cpp)**

**Overview**

The TestComponents class provides comprehensive testing of the game's integrated components, including data structures, user management, and game flow logic.

**Mock Implementations**

The test file implements several mock classes to simulate real components:

**MockDatabase Class**

```
class MockDatabase {
private:
    QMap<QString, QList<QString>> tables;
    QMap<QString, QString> tableSchemas;
    bool isOpen;
    QMap<QString, QString> users;
};
```

**Purpose**: Simulates database operations for user management testing.

**Key Features**:

- Table creation and schema management
- Record insertion and retrieval
- User authentication simulation
- Connection state management

**PlayerList Class**

```
struct Player {
    QString username;
    QString password;
    Player* next;
};
```

**Purpose**: Implements a linked list data structure for player management.

**Key Operations**:

- Add players (push_back)
- Insert at specific positions (insert)
- Remove players (erase)
- Search functionality (isfound, getPlayerNode)

**GameHistory Class**

```
struct Game {
    int index;
    char token;
    QString result;
    Game* next;
};
```

**Purpose**: Manages game history using a linked list structure.

**Features**:

- Automatic index management
- Game result tracking
- Dynamic insertion and deletion
- Index consistency maintenance

**Test Categories**

**1. Database Operation Tests**

**testDatabaseConnection()**

```
void testDatabaseConnection() {
    QMap<QString, QString> mockDb;
    QVERIFY2(mockDb.isEmpty(), "Database should be empty initially");

    mockDb["test_key"] = "test_value";
    QVERIFY2(!mockDb.isEmpty(), "Database should not be empty after insert");
}
```

**Purpose**: Tests basic database CRUD operations.

**Operations Tested**:

- Database initialization
- Record insertion
- Record updates

- Record deletion
- State validation

**2. Game Logic Integration Tests**

**testCheckWin()**

```
void testCheckWin() {
    QVector<QVector<char>> board(3, QVector<char>(3, '.'));

    // Test horizontal win
    board[0][0] = 'X'; board[0][1] = 'X'; board[0][2] = 'X';

    bool horizontalWin = false;
    for (int i = 0; i < 3; ++i) {
        if (board[i][0] == 'X' && board[i][1] == 'X' && board[i][2] == 'X') {
            horizontalWin = true;
            break;
        }
    }
    QVERIFY(horizontalWin);
}
```

**Purpose**: Tests win detection algorithms integrated with board representation.

**Win Types Tested**:

- Horizontal wins (all rows)
- Vertical wins (all columns)
- Diagonal wins (both diagonals)

**3. Data Structure Tests**

**testPlayerList()**

```
void testPlayerList() {
```

```
    PlayerList players;

    // Test empty list
    QVERIFY(players.empty());
    QCOMPARE(players.getSize(), 0);

    // Test adding players
    players.push_back("player1", "pass1");
    QVERIFY(!players.empty());
    QCOMPARE(players.getSize(), 1);
}
```

**Purpose**: Validates linked list implementation for player management.

**Operations Tested**:

- List initialization
- Element insertion (back and at position)
- Element removal
- Search operations
- Size management
- Node retrieval

**testGameHistory()**

**Purpose**: Tests game history linked list implementation.

**Key Features Tested**:

- Automatic index assignment
- Insertion at specific positions
- Index updating after modifications
- Memory management
- Node access methods

**4. Game Flow Tests**

**testResetGame()**

```
void testResetGame() {
    QVector<QVector<char>> board(3, QVector<char>(3, '.'));
    int movesLeft = 5;

    // Set some moves
    board[0][0] = 'X';
    board[1][1] = 'O';

    // Reset the game
    board = QVector<QVector<char>>(3, QVector<char>(3, '.'));
    movesLeft = 9;

    // Verify reset
    for(int i = 0; i < 3; ++i) {
        for(int j = 0; j < 3; ++j) {
            QCOMPARE(board[i][j], '.');
        }
    }
}
```

**Purpose**: Tests game reset functionality.

**Validations**:

- Board cleared to initial state
- Move counter reset
- Game state consistency

**testOnClick()**

**Purpose**: Tests move validation and execution.

**Scenarios**:

- Valid moves on empty cells
- Invalid moves on occupied cells
- Out-of-bounds move attempts
- Move counter updates
- Player token assignment

**5. User Management Tests**

**testSignUp()**

```
void testSignUp() {
    MockDatabase db;
    QVERIFY(db.open());

    // Test successful sign up
    QVERIFY(db.addUser("testuser", "testpass"));
    QVERIFY(db.verifyUser("testuser", "testpass"));

    // Test duplicate username
    QVERIFY(!db.addUser("testuser", "differentpass"));
}
```

**Purpose**: Tests user registration functionality.

**Test Cases**:

- Successful user creation
- Duplicate username prevention
- Empty field validation
- Special character handling
- Password verification

**testSignIn()**

**Purpose**: Tests user authentication.

**Scenarios**:

- Valid credentials
- Invalid passwords
- Non-existent users
- Case sensitivity
- Multiple user sessions

**testAccountManagement()**

**Purpose**: Tests comprehensive account management.

**Features Tested**:

- Multiple account creation
- Account verification
- Password updates
- Cross-account validation
- Account persistence

**6. Edge Case and Error Handling Tests**

**testPlayerListEdgeCases()**

**Purpose**: Tests boundary conditions for player list operations.

**Edge Cases**:

- Invalid insertion positions
- Empty list operations
- Out-of-bounds access
- Memory management
- Error state handling

**testGameHistoryEdgeCases()**

**Purpose**: Tests boundary conditions for game history operations.

**Edge Cases**:

- Invalid indices
- Empty history operations
- Index consistency
- Memory cleanup
- Error recovery

---

**Best Practices Demonstrated**

**1. Test Organization**

- **Logical Grouping**: Tests are organized by functionality
- **Clear Naming**: Descriptive test method names
- **Comprehensive Coverage**: Both positive and negative test cases

**2. Test Isolation**

- **Independent Tests**: Each test method is self-contained
- **Setup/Cleanup**: Proper resource management
- **Mock Objects**: Isolated testing without external dependencies

**3. Assertion Quality**

- **Specific Assertions**: Use of QVERIFY and QCOMPARE appropriately
- **Meaningful Messages**: Clear failure messages with QVERIFY2
- **Multiple Validations**: Comprehensive state checking

**4. Edge Case Testing**

- **Boundary Conditions**: Testing limits and edge cases
- **Error Scenarios**: Invalid input handling
- **State Consistency**: Invariant validation

**5. Code Quality**

- **Clean Code**: Well-structured and readable test code
- **Documentation**: Clear comments explaining test purposes

- **Maintainability**: Easy to extend and modify

---

**Running the Tests**

**Prerequisites**

- Qt development environment
- QTest framework
- CMake or qmake build system

**Compilation**

```
# Using qmake
qmake test_ai.pro
make
./test_ai

# Using CMake
cmake .
make
./test_ai
./test_components
```

**Expected Output**

```
********* Start testing of TestGameLogic *********
Config: Using QtTest library 5.x.x
PASS   : TestGameLogic::initTestCase()
PASS   : TestGameLogic::testEmptyBoard()
PASS   : TestGameLogic::testHorizontalWinConditions()
...
PASS   : TestGameLogic::cleanupTestCase()
```

Totals: X passed, 0 failed, 0 skipped, 0 blacklisted, Xms

********* Finished testing of TestGameLogic *********

**Continuous Integration**

These tests can be integrated into CI/CD pipelines for automated testing:

```
# Example GitHub Actions workflow
name: Run Tests
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Setup Qt
      uses: jurplel/install-qt-action@v2
    - name: Build and Test
      run: |
        qmake
        make
        ./test_ai
        ./test_components
```

---

**Conclusion**

This comprehensive testing suite demonstrates professional-grade testing practices for a Tic-Tac-Toe game application. The tests provide:

**Key Strengths**

1. **Complete Coverage**: All major game components are tested
2. **Robust Validation**: Both positive and negative test cases

3. **Professional Structure**: Well-organized and maintainable code
4. **Edge Case Handling**: Comprehensive boundary testing
5. **Integration Testing**: Components tested both individually and together

**Testing Value**

- **Quality Assurance**: Ensures game logic correctness
- **Regression Prevention**: Catches bugs introduced by changes
- **Documentation**: Tests serve as living documentation
- **Confidence**: Provides confidence in code reliability
- **Maintenance**: Facilitates safe refactoring and updates

**Future Enhancements**

1. **Performance Tests**: Add timing and performance benchmarks
2. **UI Testing**: Include Qt GUI testing for user interface
3. **Load Testing**: Test with multiple concurrent users
4. **Property-Based Testing**: Add randomized test generation
5. **Coverage Metrics**: Implement code coverage measurement

This testing suite serves as an excellent foundation for maintaining and extending the Tic-Tac-Toe game application while ensuring high code quality and reliability.