

Tic-Tac-Toe QTest Integrated Testing Documentation

Introduction

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

- **test_components.cpp**): Focused on testing all integrated components of game
-

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()

-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 QTest 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