# Tic Tac Toe Unit Test Documentation

## Table of Contents



## Overview

This document provides comprehensive documentation for the unit test suite implemented for a Tic Tac Toe game using the Qt Test framework (QTest). The test suite validates the core game logic functionality, ensuring reliability and correctness of the game implementation.

### Project Files

- **test_gomoku.h**: Header file containing test class declaration
- **test_gomoku.cpp**: Implementation file with all test cases
- **Target Class**: GameLogic (the main game logic class being tested)

## Test Framework

### Qt Test Framework (QTest)

The test utilizes Qt's built-in testing framework, which provides:

- **Assertion Macros**: QCOMPARE, QVERIFY for validation
- **Test Organization**: Automatic test discovery and execution
- **Signal Testing**: QSignalSpy for testing Qt signals
- **Setup/Teardown**: Lifecycle methods for test preparation and cleanup

### Test Class Structure

```
class TestGomoku : public QObject
{
    Q_OBJECT
private slots:
    void initTestCase();      // Global setup
    void cleanupTestCase();   // Global cleanup
    void init();              // Per-test setup
    void cleanup();           // Per-test cleanup

    // Test methods
    void testNewGame();
    // ... other test methods
};
```

## Test Structure

### Lifecycle Management

**Setup and Teardown Methods**

1. **initTestCase()**: Called once before all tests (currently empty)
2. **cleanupTestCase()**: Called once after all tests (currently empty)
3. **init()**: Called before each individual test
   - Creates a fresh GameLogic instance
   - Ensures test isolation
4. **cleanup()**: Called after each individual test
   - Deletes the GameLogic instance
   - Prevents memory leaks

### Test Isolation

Each test method operates on a fresh GameLogic instance, ensuring:

- No state contamination between tests
- Reliable and repeatable test results
- Independent test execution

## Test Categories

### 1. Game Initialization Tests

- **Purpose**: Verify proper game setup
- **Coverage**: Initial state validation
- **Test Methods**: `testNewGame()`

### 2. Game Mechanics Tests

- **Purpose**: Validate core gameplay functionality
- **Coverage**: Move validation, player turns, board state
- **Test Methods**: `testMakeMove()`, `testMakeMoveInvalid()`, `testCurrentPlayer()`

### 3. Game Logic Tests

- **Purpose**: Test win/loss/tie conditions
- **Coverage**: Complete game scenarios
- **Test Methods**: `testWinCondition()`, `testTieGame()`, `testGameOver()`

### 4. AI Integration Tests

- **Purpose**: Validate AI opponent functionality
- **Coverage**: AI moves, difficulty settings
- **Test Methods**: `testAIMove()`, `testAIDifficulty()`

### 5. Game Features Tests

- **Purpose**: Test additional game features
- **Coverage**: Replay functionality, serialization
- **Test Methods**: `testReplay()`, `testGameSerialization()`

## Individual Test Cases

### testNewGame()

**Purpose**: Verifies proper game initialization

**Test Scenarios**:

- Current player is set to X
- No winner initially
- Game is not over
- AI mode is currently set
- All board cells are empty

**Assertions**:

```
QCOMPARE(gameLogic->getCurrentPlayer(), 'X');
QVERIFY(gameLogic->getWinner() == '\0' || gameLogic->getWinner() == ' ');
QVERIFY(gameLogic->isGameOver() == false);
QVERIFY(gameLogic->isAIMode() == false);
```

### testMakeMove()

**Purpose**: Tests valid move execution

**Test Scenarios**:

- Valid moves are accepted
- Board state is updated correctly
- Player turns alternate properly

**Key Validations**:

- Move returns true for success
- Cell contains correct player marker
- Current player switches after move

### testMakeMoveInvalid()

**Purpose**: Validates move validation logic

**Test Scenarios**:

- Out of bounds coordinates rejected
- Occupied cells cannot be overwritten
- Invalid moves return false

**Edge Cases Tested**:

- Negative coordinates (-1, 0), (0, -1)
- Coordinates beyond board (3, 0), (0, 3)
- Attempting to play on occupied cell

### testWinCondition()

**Purpose**: Verifies win detection logic

**Test Implementation**:

- Creates a winning scenario (horizontal win)
- Uses QSignalSpy to monitor gameEnded signal
- Validates game state after win

**Assertions**:

- Game is marked as over
- Correct winner is identified
- gameEnded signal is emitted with correct player

### testTieGame()

**Purpose**: Tests tie/draw game detection

**Test Implementation**:

- Fills entire board without winner
- Monitors gameEnded signal
- Validates tie game state

**Key Points**:

- Game ends when board is full
- Winner remains Player.None
- Signal emitted with Player.None

### testCurrentPlayer()

**Purpose**: Validates player turn management

**Test Flow**:

1. Initial player is X
2. After X's move, current player becomes O
3. After O's move, current player returns to X

### testGameOver()

**Purpose**: Tests game termination detection

**Scenarios**:

- Game starts as not over
- After winning condition, game is over

### testAIMove()

**Purpose**: Validates AI opponent functionality

**Test Setup**:

- Enables AI mode
- Sets difficulty to Unbeatable
- Makes human move

**Validation**:

- Confirms AI mode is active
- Verifies AI automatically makes move
- Checks total occupied cells increases

### testReplay()

**Purpose**: Tests game replay functionality

**Features Tested**:

- Move history tracking
- Replay to specific move number
- Reset replay functionality

**Validations**:

- Moves count matches expected
- Board state reflects replay position
- Reset clears all moves

### testGameSerialization()

**Purpose**: Tests game state persistence

**Functionality**:

- Serializes game state to JSON
- Deserializes and validates state

**Validated Fields**:

- AI mode setting
- Difficulty level
- Move history
- Game result

### testAIDifficulty()

**Purpose**: Tests AI difficulty management

**Levels Tested**:

- Easy
- Hard
- Unbeatable

## Test Coverage Analysis

### Functional Coverage

The test suite covers:

✅ **Core Game Mechanics**
- Game initialization
- Move validation
- Player turn management
- Board state management

✅ **Game Logic**
- Win condition detection
- Tie game detection
- Game termination

✅ **AI Features**
- Move generation
- Difficulty settings
- Human vs AI gameplay

✅ **Advanced Features**
- Game replay system
- JSON serialization/deserialization

### Coverage Gaps

Areas that could benefit from additional testing:

⚠️ **UI Strategy Testing**
- Specific AI move quality validation
- Difficulty-specific behavior verification

⚠️ **Edge Case Scenarios**
- Rapid successive moves
- State transitions during replay

⚠️ **Error Handling**
- Invalid JSON loading
- Corrupted game state recovery

## Best Practices Implemented

### 1. Test Organization

- Clear test method naming convention
- Logical grouping of related tests
- Comprehensive coverage of public API

### 2. Test Independence

- Fresh instance for each test
- Proper setup and teardown
- No shared state between tests

### 3. Signal Testing

- Proper use of QSignalSpy
- Validation of signal parameters
- Event-driven behavior testing

### 4. Assertion Quality

- Specific comparisons using QCOMPARE
- Boolean validations with QVERIFY
- Clear expected vs actual values

### 5. Test Data Management

- Systematic board state checking
- Comprehensive move validation
- State transition verification

## Recommendations

### Immediate Improvements

**1. Add More Win Condition Tests**

- Vertical wins
- Diagonal wins
- All possible winning combinations

**2. Enhance AI Testing**

- Test AI move quality at different difficulties
- Validate AI doesn't make invalid moves
- Test AI response time

**3. Add Performance Tests**

- Large number of games simulation
- Memory usage validation
- Response time benchmarks

**4. Integration Tests**

- UI interaction simulation
- End-to-end game scenarios
- Multi-game session testing

**5. Stress Testing**

- Rapid move sequences
- Memory leak detection
- Resource usage monitoring

### Documentation Improvements

**1. Test Case Documentation**

- Add inline comments explaining complex test scenarios
- Document expected behaviors
- Include failure case explanations

**2. Test Data Documentation**

- Document test board configurations
- Explain move sequences
- Clarify expected outcomes

## Conclusion

The current test suite provides solid coverage of the Tic Tac Toe game logic with structured, independent tests. The implementation follows Qt Test framework best practices and covers the major functional areas of the game.

The tests effectively validate:

- Basic game mechanics and rules
- AI opponent functionality
- Advanced features like replay and serialization
- Error conditions and edge cases

This comprehensive test suite serves as both validation for the current implementation and documentation for the expected behavior of the GameLogic class, making it an invaluable asset for ongoing development and maintenance.