**Tic-Tac-Toe Game Testing Documentation**

**Testing Overview**

This document provides comprehensive documentation of the testing strategies, methodologies, and results for the Tic-Tac-Toe game application. The testing suite ensures robust functionality across all game components including AI logic, user authentication, game mechanics, data persistence, and system integration.

**Testing Objectives**

* Functionality Verification: Ensure all features work as specified
* Performance Validation: Verify acceptable response times and resource usage
* Data Integrity: Confirm proper data handling and persistence
* Integration Reliability: Validate component interactions
* Edge Case Handling: Test boundary conditions and error scenarios

**Testing Framework**

Google Test (gtest)

The project utilizes Google Test framework for comprehensive C++ testing capabilities:

#include <gtest/gtest.h>

Key Features Used:

* Test fixtures with SetUp() and TearDown() methods
* Parameterized tests for comprehensive scenario coverage
* Performance timing with <chrono> integration
* Assertion macros for precise validation
* Test organization with test suites and cases

Testing Environment Setup

* File Management: Automatic cleanup of test databases and history files
* Isolation: Each test runs in a clean environment
* Resource Tracking: Systematic management of test artifacts

**Test Suite Architecture**

Test File Organization

1. test\_game.cpp - Core game logic tests
2. test\_ai.cpp - AI algorithm and strategy tests
3. test\_auth.cpp - User authentication and security tests
4. test\_history.cpp - Data persistence and file I/O tests
5. test\_integration.cpp - System integration and workflow tests

**Test Class Structure**

class ComponentTest : public ::testing::Test {

protected:

void SetUp() override {

// Initialize test environment

// Clean up existing test files

}

void TearDown() override {

// Cleanup test artifacts

// Reset system state

}

};

**Unit Tests**

**Game Logic Tests (test\_game.cpp)**

**Test Coverage: 11 test cases**

**Core Functionality Tests**

* Initialization Test: Validates initial game state
  + Current player is X
  + No winner initially
  + All positions empty
  + 9 available moves
* **Basic Move Tests: Verifies move mechanics**
  + Valid move placement
  + Player alternation
  + Invalid move rejection
  + Available moves tracking

**Win Condition Tests**

* Horizontal Win: Tests row-based victories
* Vertical Win: Tests column-based victories
* Diagonal Win (Main): Tests top-left to bottom-right diagonal
* Diagonal Win (Anti): Tests top-right to bottom-left diagonal

**Game State Tests**

* Draw Detection: Validates tie game scenarios
* Available Moves: Confirms accurate move tracking
* Player Alternation: Ensures proper turn management

**AI Logic Tests (test\_ai.cpp)**

**Test Coverage: 9 test cases**

**Strategic Intelligence Tests**

* **Winning Move Detection: AI identifies immediate winning opportunities**
  + Horizontal winning moves
  + Vertical winning moves
  + Diagonal winning moves
* **Strategic Positioning: AI follows optimal strategies**
  + Takes center when available
  + Takes corners when opponent has center
  + Prioritizes immediate wins over defensive moves

**Validation Tests**

* Move Validity: Ensures AI only suggests legal moves
* Performance: AI responds within acceptable time limits (<1 second)
* AI vs AI: Validates optimal play results in draws

**Authentication Tests (test\_auth.cpp)**

**Test Coverage: 5 test cases**

**User Management Tests**

* **Registration: New user creation with validation**
  + Successful registration with valid credentials
  + Duplicate username rejection
  + Empty field validation
  + Special character handling
* **Login Authentication: Credential verification**
  + Correct password acceptance
  + Wrong password rejection
  + Non-existent user handling

**Data Security Tests**

* Multiple Users: Independent user account management
* Data Persistence: User data survives application restarts
* Special Characters: Handles complex passwords and usernames

**History Management Tests (test\_history.cpp)**

**Test Coverage: 7 test cases**

**Data Persistence Tests**

* File Operations: Save and load game results
* Multiple Results: Handle sequential game storage
* Empty History: Graceful handling of no prior games
* Round-trip Integrity: Data consistency across save/load cycles

**Performance Tests**

* **Large History Files: Handles 1000+ game records efficiently** 
  + Save performance: <5 seconds for 1000 entries
  + Load performance: <1 second for 1000 entries

**Integration Tests**

**System Integration Tests (test\_integration.cpp)**

**Test Coverage: 6 comprehensive test cases**

**Complete User Session Test**

**Validates end-to-end user workflow:**

1. User Registration → 2. User Login → 3. Game Play → 4. Result Storage → 5. History Verification

**Multi-Player Game Session**

**Tests multiple users playing games simultaneously:**

* Independent user sessions
* Concurrent game state management
* Consistent result tracking
* Cross-user data isolation

**AI Tournament Testing**

**Automated AI vs AI competitions:**

* Multiple game scenarios
* Performance consistency
* Optimal play validation (all games result in draws)

**Stress Testing**

**High-load system validation:**

* 10 users × 5 games each = 50 total operations
* Performance requirement: <10 seconds total
* Data integrity across all operations

**Test Coverage Analysis**

**Component Coverage Summary**

| **Component** | **Test Cases** | **Coverage Areas** | **Critical Paths** |
| --- | --- | --- | --- |
| **Game Logic** | 11 | Move validation, Win detection, State management | All win conditions, Draw scenarios |
| **AI Strategy** | 9 | Decision making, Performance, Strategy validation | Winning moves, Optimal play |
| **Authentication** | 5 | User management, Security, Data persistence | Login/Register workflows |
| **History** | 7 | File I/O, Data integrity, Performance | Save/Load operations |
| **Integration** | 6 | System workflows, Multi-user scenarios | Complete user sessions |

**Critical Path Testing**

* Game Win Detection: 100% coverage of all win conditions
* AI Decision Making: All strategic scenarios tested
* Data Persistence: Complete save/load cycle validation
* User Authentication: All security scenarios covered

**Performance Testing**

**AI Performance Requirements**

TEST\_F(AITest, Performance) {

auto start = std::chrono::high\_resolution\_clock::now();

auto move = ai.findBestMove(game);

auto end = std::chrono::high\_resolution\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);

EXPECT\_LT(duration.count(), 1000); // Less than 1 second

}

**Data Operations Performance**

* **History File Operations:** 
  + Save 1000 entries: <5 seconds
  + Load 1000 entries: <1 second
* **Authentication Operations: Instantaneous response**
* **Game State Updates: Real-time performance**

**Stress Test Results**

* 10 Users × 5 Games: Completed in <10 seconds
* Memory Usage: Stable across extended operations
* File System: No corruption under high load

**Test Results**

Test Execution Summary

[==========] Running 37 tests from 5 test suites.

[----------] Global test environment set-up**.**

[----------] 11 tests from GameTest

[----------] 9 tests from AITest

[----------] 5 tests from AuthTest

[----------] 7 tests from HistoryTest

[----------] 6 tests from IntegrationTest

[----------] Global test environment tear-down

[==========] 37 tests from 5 test suites ran. (1185 ms total)

[ PASSED ] 37 tests.

**Success Metrics**

* Test Pass Rate: 100% (37/37 tests passed)
* Total Execution Time: 1.185 seconds
* Performance Compliance: All timing requirements met
* Memory Management: No memory leaks detected

**Key Test Achievements**

1. Zero Failures: All implemented functionality works correctly
2. Performance Targets Met: AI responses <1s, bulk operations <10s
3. Data Integrity: 100% accuracy in save/load operations
4. Multi-user Support: Confirmed isolated user experiences

**Testing Best Practices**

**Test Design Principles**

1. Test Isolation: Each test runs independently
2. Clean Environment: Automatic setup/teardown
3. Comprehensive Coverage: All code paths tested
4. Performance Validation: Timing requirements enforced
5. **Edge Case Testing: Boundary conditions verified**

**Code Quality Assurance**

// Example of robust test structure

TEST\_F(ComponentTest, SpecificScenario) {

// Arrange: Set up test conditions

// Act: Execute the functionality

// Assert: Verify expected outcomes

// Performance: Validate timing if applicable

}

File Management Strategy

void TearDown() override {

// Clean up test files systematically

cleanupTestFiles();

}

void cleanupTestFiles() {

for (const auto& filename : testFiles) {

if (std::filesystem::exists(filename)) {

std::filesystem::remove(filename);

}

}

}

**Continuous Testing Strategy**

**Automated Test Execution**

* Pre-commit Hooks: Run core tests before code commits
* Build Integration: Full test suite execution on builds
* Performance Monitoring: Track timing trends over time

**Test Maintenance**

* Regular Review: Quarterly test case assessment
* Coverage Analysis: Identify and fill testing gaps
* Performance Benchmarking: Monitor and maintain timing requirements

**Quality Gates**

* 100% Test Pass Rate: No failing tests allowed in main branch
* Performance Compliance: All timing requirements must be met
* Code Coverage: Maintain high coverage across all components

**Conclusion**

**The comprehensive test suite provides robust validation of the Tic-Tac-Toe game system through:**

* 37 Test Cases covering all system components
* 100% Pass Rate ensuring reliable functionality
* Performance Validation confirming responsive user experience
* Integration Testing validating complete user workflows
* Stress Testing ensuring system stability under load

The testing framework successfully validates that the application meets all functional requirements while maintaining excellent performance characteristics and data integrity across all supported operations.