Testing Documentation

1. Testing Strategy

1.1 Testing Approach

The Tic-Tac-Toe project employs a comprehensive testing strategy using Google Test framework to ensure software quality and reliability across all components:

- **Unit Testing**: Individual component testing for game logic, user authentication, and AI algorithms
- **Integration Testing**: Component interaction testing between database, authentication, and game systems

1.2 Test Coverage Goals

- **Minimum 85% code coverage** across all modules
- **100% coverage for critical paths**: Authentication, game logic, and AI decision-making
- **All public API methods tested** with both valid and invalid inputs
- **Edge case coverage** for boundary conditions and error scenarios

2. Unit Test Cases

2.1 User Authentication Testing Documentation

2.1.1 Database Component Tests

The database component handles user data persistence and retrieval operations[1].

Test Case: AddAndCheckUser

```
TEST_F(DatabaseTest, AddAndCheckUser) {
  EXPECT_TRUE(db->addUser("testuser", "hashedpass"));
  EXPECT_TRUE(db->userExists("testuser"));
  EXPECT_FALSE(db->userExists("nonexistent"));
}
```

- **Purpose**: Validates user creation and existence checking
- **Expected Outcome**: Successfully adds user and correctly identifies existing/non-existing users

- **Critical Path**: User registration workflow

Test Case: GetUserPassword

```
TEST_F(DatabaseTest, GetUserPassword) {
  db->addUser("testuser", "hashedpass");
  std::string password;
  EXPECT_TRUE(db->getUserPassword("testuser", password));
  EXPECT_EQ(password, "hashedpass");
  EXPECT_FALSE(db->getUserPassword("nonexistent", password));
}
```

- **Purpose**: Verifies password retrieval functionality
- **Expected Outcome**: Returns correct password for existing users, fails for non-existing users
- **Security Consideration**: Ensures password hash integrity

Test Case: AddDuplicateUserFails

```
TEST_F(DatabaseTest, AddDuplicateUserFails) {
  EXPECT_TRUE(db->addUser("dupuser", "pass1"));
  EXPECT_FALSE(db->addUser("dupuser", "pass2"));
}
```

- **Purpose**: Prevents duplicate user registration
- **Expected Outcome**: First registration succeeds, duplicate fails
- **Data Integrity**: Maintains unique usernames

Test Case: CaseSensitivityOfUsernames

```
cpp
TEST_F(DatabaseTest, CaseSensitivityOfUsernames) {
  EXPECT_TRUE(db->addUser("CaseUser", "pass"));
  EXPECT_FALSE(db->userExists("caseuser"));
}
```

- **Purpose**: Validates case-sensitive username handling
- **Expected Outcome**: Different cases treated as different users
- **Security Feature**: Prevents username confusion attacks

2.1.2 User Authentication Tests

The authentication system manages user login, registration, and session security.

Test Case: RegisterNewUser

```
TEST_F(UserAuthTest, RegisterNewUser) {
   EXPECT_TRUE(auth->registerUser("testuser", "testpass1"));
   EXPECT_FALSE(auth->registerUser("testuser", "testpass2"));
}
```

- **Purpose**: Tests user registration workflow
- **Expected Outcome**: New user registration succeeds, duplicate fails
- **Authentication Flow**: Foundation for secure user management

Test Case: LoginSuccess

```
TEST_F(UserAuthTest, LoginSuccess) {
   EXPECT_TRUE(auth->registerUser("testuser", "testpass1"));
   EXPECT_TRUE(auth->login("testuser", "testpass1"));
}
```

- **Purpose**: Validates successful authentication
- **Expected Outcome**: Correct credentials allow login
- **Security Check**: Password verification works correctly

Test Case: LoginWrongPassword

```
TEST_F(UserAuthTest, LoginWrongPassword) {
   EXPECT_TRUE(auth->registerUser("testuser", "testpass1"));
   EXPECT_FALSE(auth->login("testuser", "wrongpass"));
}
```

- **Purpose**: Ensures incorrect passwords are rejected
- **Expected Outcome**: Login fails with wrong password
- **Security Feature**: Prevents unauthorized access

Test Case: SpecialCharacterCredentials

```
TEST_F(UserAuthTest, SpecialCharacterCredentials) {
   EXPECT_TRUE(auth->registerUser("user!@#", "p@ss!3"));
   EXPECT_TRUE(auth->login("user!@#", "p@ss!3"));
}
```

- **Purpose**: Tests handling of special characters in credentials
- **Expected Outcome**: Special characters handled correctly
- **Robustness**: Supports complex passwords

2.2 Game Components Testing Documentation (AI and Board)

2.2.1 AI Algorithm Tests

The AI component implements strategic decision-making using minimax algorithm[1].

Test Case: ImmediateWinXRow

```
TEST(AITest, ImmediateWinXRow) {
   Board board;
   board.makeMove(0, 0, Player::X);
   board.makeMove(1, 0, Player::O);
   board.makeMove(0, 1, Player::X);
   board.makeMove(1, 1, Player::O);
   auto move = findBestMove(board, Player::X);
   EXPECT_EQ(move, std::make_pair(0, 2));
}
```

- **Purpose**: Validates AI recognizes immediate winning opportunities
- **Expected Outcome**: AI chooses position (0,2) to complete winning row
- **Strategic Intelligence**: Priority 1 - Take winning move

Test Case: BlockOpponentX

```
TEST(AITest, BlockOpponentX) {
   Board board;
   board.makeMove(0, 0, Player::X);
```

```
    board.makeMove(1, 1, Player::O);
    board.makeMove(2, 2, Player::X);
    board.makeMove(2, 1, Player::O);
    auto move = findBestMove(board, Player::X);
    EXPECT_EQ(move, std::make_pair(0, 1));
}
```

- **Purpose**: Tests AI's defensive blocking capability
- **Expected Outcome**: AI blocks opponent's winning column at (0,1)
- **Strategic Intelligence**: Priority 2 - Block opponent wins

Test Case: CenterMoveX

```
TEST(AITest, CenterMoveX) {
    Board board;
    auto move = findBestMove(board, Player::X);
    EXPECT_EQ(move, std::make_pair(1, 1));
}
```

- **Purpose**: Validates optimal opening move strategy
- **Expected Outcome**: AI takes center position on empty board
- **Strategic Intelligence**: Center position provides maximum opportunities

Test Case: PreventForkO

```
TEST(AITest, PreventForkO) {
    Board board;
    board.makeMove(0, 0, Player::X);
    board.makeMove(1, 1, Player::O);
    board.makeMove(2, 2, Player::X);
    auto move = findBestMove(board, Player::O);
    std::vector<std::pair<int, int>> edges = {{0, 1}, {1, 0}, {1, 2}, {2, 1}};
    bool is_edge_move = std::any_of(edges.begin(), edges.end(),
        [&move](const auto& edge) {
            return edge.first == move.first && edge.second == move.second;
```

```
    });
  EXPECT_TRUE(is_edge_move);
}
```

- **Purpose**: Tests advanced fork prevention strategy
- **Expected Outcome**: AI plays edge move to prevent opponent's fork setup
- **Strategic Intelligence**: Advanced tactical awareness

2.2.2 Board Logic Tests

The board component manages game state, move validation, and win detection[1].

Test Case: ValidMoveInsideGrid

```
TEST(BoardTest, ValidMoveInsideGrid) {
  Board board;
  EXPECT_TRUE(board.isValidMove(0, 0));
}
```

- **Purpose**: Validates move validation for legal positions
- **Expected Outcome**: Positions within 3x3 grid are valid
- **Game Rules**: Enforces board boundaries

Test Case: InvalidMoveOutOfBounds

```
TEST(BoardTest, InvalidMoveOutOfBounds) {
  Board board;
  EXPECT_FALSE(board.isValidMove(-1, 0));
  EXPECT_FALSE(board.isValidMove(3, 1));
  EXPECT_FALSE(board.isValidMove(0, 3));
}
```

- **Purpose**: Rejects moves outside game board
- **Expected Outcome**: Negative indices and indices ≥3 are invalid
- **Error Prevention**: Prevents array bounds violations

Test Case: InvalidMoveOnOccupiedCell

```
TEST(BoardTest, InvalidMoveOnOccupiedCell) {
  Board board;
```

```
  board.makeMove(0, 0, Player::X);
  EXPECT_FALSE(board.isValidMove(0, 0));
}
```

- **Purpose**: Prevents overwriting existing moves
- **Expected Outcome**: Occupied cells cannot be played again
- **Game Rules**: Enforces move permanence

Test Case: CheckWinnerRow

```
TEST(BoardTest, CheckWinnerRow) {
  Board board;
  board.makeMove(1, 0, Player::O);
  board.makeMove(1, 1, Player::O);
  board.makeMove(1, 2, Player::O);
  WinInfo win = board.checkWinner();
  EXPECT_EQ(win.winner, Player::O);
  EXPECT_EQ(win.type, "row");
  EXPECT_EQ(win.index, 1);
}
```

- **Purpose**: Detects horizontal winning conditions
- **Expected Outcome**: Identifies Player::O wins row 1 with correct metadata
- **Win Detection**: Core game logic for victory conditions

Test Case: CheckWinnerMainDiagonal

```
TEST(BoardTest, CheckWinnerMainDiagonal) {
  Board board;
  board.makeMove(0, 0, Player::X);
  board.makeMove(1, 1, Player::X);
  board.makeMove(2, 2, Player::X);
  WinInfo win = board.checkWinner();
  EXPECT_EQ(win.winner, Player::X);
  EXPECT_EQ(win.type, "diag");
}
```

- **Purpose**: Validates diagonal win detection
- **Expected Outcome**: Recognizes main diagonal victory
- **Win Detection**: Comprehensive victory condition coverage

Test Case: BoardIsFullAfterAllMoves

```
TEST(BoardTest, BoardIsFullAfterAllMoves) {
  Board board;
  Player p = Player::X;
  for (int i = 0; i < 3; ++i)
    for (int j = 0; j < 3; ++j)
      board.makeMove(i, j, p);
  EXPECT_TRUE(board.isFull());
}
```

- **Purpose**: Detects board full condition for tie games
- **Expected Outcome**: Returns true when all 9 positions occupied
- **Game State**: Essential for tie detection

2.3 Game History Testing Documentation

2.3.1 Game History Tests

The game history system manages persistent storage of game sessions and player statistics[1].

Test Case: InitializeGame

```
TEST_F(GameHistoryTest, InitializeGame) {
  int game_id = history->initializeGame(1, 2);
  EXPECT_GT(game_id, 0);
  auto game = history->getGameById(game_id);
  EXPECT_EQ(game.id, game_id);
  EXPECT_EQ(game.playerX_id.value(), 1);
  EXPECT_EQ(game.playerO_id.value(), 2);
  EXPECT_FALSE(game.winner_id.has_value());
}
```

- **Purpose**: Tests game session creation and retrieval
- **Expected Outcome**: Creates game with valid ID and correct player assignments
- **Data Persistence**: Foundation for game history tracking

Test Case: RecordMoves

```
TEST_F(GameHistoryTest, RecordMoves) {
    int game_id = history->initializeGame(1, 2);
    EXPECT_TRUE(history->recordMove(game_id, 4)); // Center
    EXPECT_TRUE(history->recordMove(game_id, 0)); // Top-left
    EXPECT_TRUE(history->recordMove(game_id, 8)); // Bottom-right
    auto game = history->getGameById(game_id);
    EXPECT_EQ(game.moves.size(), 3);
    EXPECT_EQ(game.moves[0].position, 4);
}
```

- **Purpose**: Validates move recording and sequence preservation
- **Expected Outcome**: Moves stored in correct order with accurate positions
- **Game Replay**: Enables game analysis and replay functionality

Test Case: SetWinner

```
TEST_F(GameHistoryTest, SetWinner) {
    int game_id = history->initializeGame(1, 2);
    history->recordMove(game_id, 4);
    history->recordMove(game_id, 0);
    EXPECT_TRUE(history->isGameActive(game_id));
    EXPECT_TRUE(history->setWinner(game_id, 1));
    auto game = history->getGameById(game_id);
    EXPECT_EQ(game.winner_id.value(), 1);
    EXPECT_FALSE(history->isGameActive(game_id));
}
```

- **Purpose**: Tests game completion and winner assignment
- **Expected Outcome**: Winner recorded correctly, game marked inactive
- **Game State Management**: Proper session lifecycle handling

Test Case: AIGame

```
TEST_F(GameHistoryTest, AIGame) {
    int game_id = history->initializeGame(1, std::nullopt);
    history->recordMove(game_id, 0);
    history->recordMove(game_id, 4);
    EXPECT_TRUE(history->setWinner(game_id, -2));
    auto game = history->getGameById(game_id);
    EXPECT_EQ(game.winner_id.value(), -2);
    EXPECT_FALSE(game.playerO_id.has_value());
}
```

- **Purpose**: Tests AI game handling with special winner codes
- **Expected Outcome**: AI games recorded with player vs null opponent
- **Game Modes**: Supports both PvP and PvAI game types

2.4 GUI Testing Documentation

2.4.1 Login Page Testing

Test Case: InitialState

```
TEST_F(LoginPageTest, InitialState) {
    QLineEdit* usernameEdit = loginPage->findChild<QLineEdit*>("m_usernameEdit");
    EXPECT_TRUE(usernameEdit->text().isEmpty());
    QLineEdit* passwordEdit = loginPage->findChild<QLineEdit*>("m_passwordEdit");
    EXPECT_TRUE(passwordEdit->text().isEmpty());
}
```

- **Purpose**: Verifies that the login page initializes with empty input fields and status label1
- **Expected Outcome**: Username field is empty, Password field is empty, Status label contains no text

Test Case: SuccessfulLogin

```
TEST_F(LoginPageTest, SuccessfulLogin) {
    EXPECT_CALL(*mockAuth, login(testing::_, testing::_))
```

```
    .WillOnce(testing::Return(true));
  QSignalSpy spy(loginPage, SIGNAL(loginSuccessful(const QString&)));
  usernameEdit->setText("testuser");
  passwordEdit->setText("password123");
  QTest::mouseClick(loginButton, Qt::LeftButton);
  ASSERT_EQ(spy.count(), 1);
}
```

- **Purpose**: Tests successful user authentication and signal emission[1]
- **Expected Outcome**: Authentication service called with correct credentials, loginSuccessful signal emitted with username

Test Case: FailedLogin

```
TEST_F(LoginPageTest, FailedLogin) {
 // Set up the mock to return false for login
  EXPECT_CALL(*mockAuth, login(testing::_, testing::_))
    .WillOnce(testing::Return(false));
  // Set up signal spy to ensure the loginSuccessful signal is NOT emitted
  QSignalSpy spy(loginPage, SIGNAL(loginSuccessful(const QString&)));
  // Fill in the login form
  QLineEdit* usernameEdit = loginPage->findChild<QLineEdit*>("m_usernameEdit");
  QLineEdit* passwordEdit = loginPage->findChild<QLineEdit*>("m_passwordEdit");
  QPushButton* loginButton = loginPage->findChild<QPushButton*>("m_loginButton");
  ASSERT_NE(loginButton, nullptr);
  usernameEdit->setText("testuser");
  passwordEdit->setText("wrongpassword");
  // Click the login button
  QTest::mouseClick(loginButton, Qt::LeftButton);
  // Check that the signal was NOT emitted
  EXPECT_EQ(spy.count(), 0);
  // Check that the status label shows an error
  QLabel* statusLabel = loginPage->findChild<QLabel*>("m_statusLabel");
  EXPECT_FALSE(statusLabel->text().isEmpty());
```

```
    EXPECT_TRUE(statusLabel->text().contains("failed", Qt::CaseInsensitive));
}
```

- **Purpose**: Verifies proper handling of authentication failures[1]
- **Expected Outcome**: No loginSuccessful signal emitted, Error message displayed in status label

2.4.2 Game Window Testing

Test Case: InitialState

```
TEST_F(GameWindowTest, InitialState) {
// Check that the game window is properly initialized
    ASSERT_NE(gameWindow, nullptr);
    // Check that the window has the correct title
    EXPECT_EQ(gameWindow->windowTitle(), "Tic-Tac-Toe");
    // Check that buttons exist (without checking visibility which requires proper setup)
    QList<QPushButton*> buttons = gameWindow->findChildren<QPushButton*>();
    EXPECT_GT(buttons.size(), 0);
    // Check that some expected buttons exist
    bool foundPvPButton = false;
    bool foundPvAIButton = false;
    for (auto* btn : buttons) {
        if (btn->text() == "Player vs Player") foundPvPButton = true;
        if (btn->text() == "Player vs AI") foundPvAIButton = true;
    }
    EXPECT_TRUE(foundPvPButton);
    EXPECT_TRUE(foundPvAIButton);
}
```

- **Purpose**: Verifies initial game window state and available game modes[1]
- **Expected Outcome**: Window title set to "Tic-Tac-Toe", Player vs Player and Player vs AI buttons available

Test Case: PvPModeSelection

```
TEST_F(GameWindowTest, PvPModeSelection) {
    // Test PvP mode selection now requests second player authentication
```

```
    QSignalSpy spy(gameWindow, SIGNAL(secondPlayerAuthenticationRequested()));

    // Find and click PvP button

    QList<QPushButton*> buttons = gameWindow->findChildren<QPushButton*>();

    QPushButton* pvpBtn = nullptr;

    for (auto* btn : buttons) {

        if (btn->text() == "Player vs Player") {

            pvpBtn = btn;

            break;

        }

    }

    ASSERT_NE(pvpBtn, nullptr);

    QTest::mouseClick(pvpBtn, Qt::LeftButton);

    // Should request second player authentication

    EXPECT_EQ(spy.count(), 1);

}
```

- **Purpose**: Tests PvP mode selection and second player authentication request[1]
- **Expected Outcome**: secondPlayerAuthenticationRequested signal emitted, Game transitions to second player login flow

2.4.3 Game History GUI Testing

Test Case: GamesListPopulation

```
TEST_F(GameHistoryGUITest, GamesListPopulation) {

    QTreeWidget* gamesTreeWidget = gameHistoryGUI->findChild<QTreeWidget*>();

    EXPECT_EQ(gamesTreeWidget->topLevelItemCount(), 3);

    QTreeWidgetItem* firstItem = gamesTreeWidget->topLevelItem(0);

    QString playersText = firstItem->text(2);

    EXPECT_TRUE(playersText.contains("vs"));

}
```

- **Purpose**: Verifies games list populates with historical game data[1]
- **Expected Outcome**: Correct number of games displayed, Game information properly formatted, Players column shows "vs" format