TIC TAC TOE PROJECT

Testing Documentation

Submitted to :
Dr/Omar Nasr

Contents

Table of figures:

Testing Documentation for Tic-Tac-Toe Application

1. Introduction

This document provides a comprehensive overview of the testing strategy, scope, and detailed test cases for the Tic-Tac-Toe application. The application, developed using Qt and C++, includes features such as user registration, authentication, game logic, and a graphical user interface. The testing efforts are focused on ensuring the reliability, functionality, and robustness of each component.

2. Test Strategy

Our testing strategy employs a multi-layered approach, combining unit testing and integration testing to validate individual components and their interactions. The primary goals are to: - Verify the correctness of core game logic. - Ensure the reliability of user management (registration and authentication). - Validate the functionality and usability of the graphical user interface. - Confirm proper data persistence and retrieval through the database.

2.1. Unit Testing

Unit tests are designed to test individual functions, methods, or classes in isolation. For this project, unit tests are implemented using the Google Test framework for C++ logic and QtTest for Qt-specific UI components. This approach allows for early detection of defects and provides a safety net for refactoring.

2.2. Integration Testing

Integration tests focus on verifying the interactions between different modules and components. For instance, testing the `MainWindow` involves interactions with the `Game` and `Database` components, often using mock objects to control dependencies and ensure test isolation where appropriate.

3. Test Environment

3.1. Development Environment

Operating System: Ubuntu

Programming Language: C++

Frameworks: Qt 6

Testing Frameworks: Google Test, QtTest

Database: SQLite (for local development and testing)

3.2. Test Data

Test data for user registration and game history is generated dynamically within the test cases to ensure isolation and repeatability. Mock objects are extensively used for database interactions to prevent side effects and speed up test execution.

4. Test Cases

4.1. `MainWindow` Test Cases

The `MainWindow` is the primary user interface for the Tic-Tac-Toe game. Its tests cover UI component rendering, user interaction, and integration with game and database functionalities. The tests are structured to validate the following aspects:

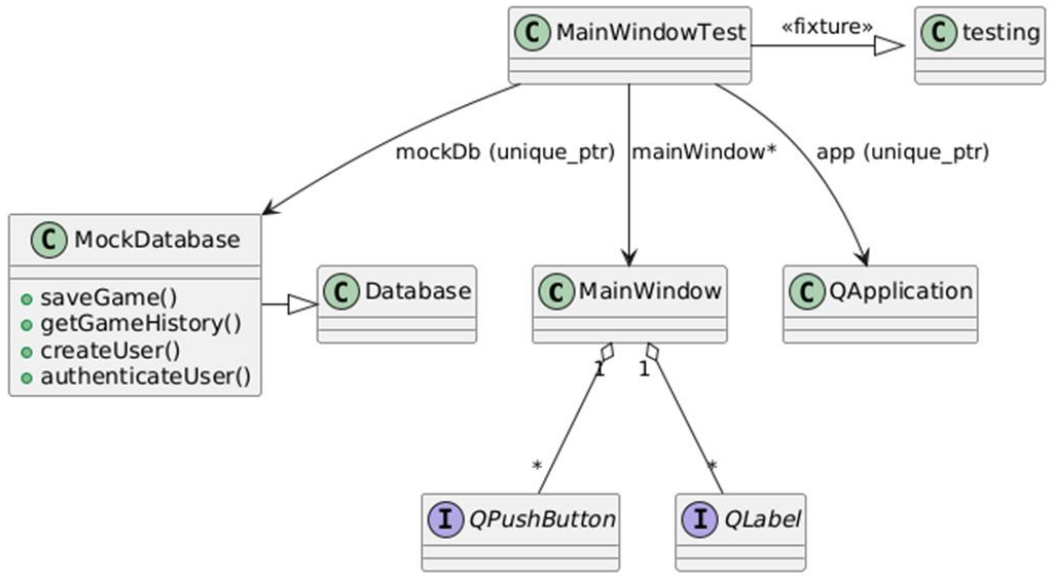

Figure 1--MainWindow test flow

4.1.1. Constructor and Initialization

Description: Verify that the `MainWindow` constructor correctly initializes all UI elements and internal states for both registered and guest users.

Preconditions: None.
Steps:

1. Create `MainWindow` instances with different user IDs (e.g., 1 for registered, -1 for guest).
2. Assert that the `MainWindow` object is not null.
3. Assert that the window is initially not visible (hidden during tests).
4. Verify basic widget properties and inheritance.

Expected Results: The `MainWindow` is successfully created, all essential components are initialized, and the window is not visible.

4.1.2. UI Component Presence and Properties

Description: Ensure that all required UI components, such as title labels, game control buttons, and player/mode indicators, are present and have appropriate properties.

Preconditions: `MainWindow` is initialized.

Steps:

1. Find all `QLabel` and `QPushButton` instances within the `MainWindow` .

2. Assert the presence of a title label containing 'TIC TAC TOE'.

3. Assert the presence of control buttons: 'VS AI', 'VS PLAYER', 'RESTART', 'HISTORY', 'LOGOUT'.

4. Assert the presence of player/turn and mode indicators. - Expected Results:

All specified UI components are present and correctly labeled.

4.1.3. Game Board Structure and Initial State

Description: Verify the structure and initial state of the game board.

Preconditions: `MainWindow` is initialized.

Steps:

1. Retrieve all game cell buttons.
2. Assert that there are exactly 9 game cells.
3. Assert that all game cells are initially empty (text is empty or a single space).
4. Assert that all game cells are enabled in test mode.
5. Verify that a `QGridLayout` is used for the game board layout.

Expected Results: The game board has 9 empty, enabled cells arranged in a grid layout.

4.1.4. User-Specific Interface Differences

Description: Confirm that the UI adapts correctly for registered and guest users, specifically regarding the visibility of the 'HISTORY' button.

Preconditions: `MainWindow` is initialized.

Steps:

1. Create `MainWindow` with a registered user ID (e.g., 1).
2. Assert that the 'HISTORY' button exists and is visible.
3. Create `MainWindow` with a guest user ID (e.g., -1).
4. Assert that the 'HISTORY' button is not visible or not present for guest users.

Expected Results: The 'HISTORY' button is visible only for registered users.

4.1.5. Button Functionality and Properties

Description: Validate that all interactive buttons have valid signals, reasonable text,

and appropriate sizing.

Preconditions: `MainWindow` is initialized.

Steps:

1. Iterate through all `QPushButton` instances in the `MainWindow`.
2. For each button, assert that its `clicked` signal is valid.
3. Assert that button text is not excessively long (e.g., < 50 characters) and is meaningful (not empty for control buttons).
4. Assert that buttons have reasonable minimum and maximum sizes.

Expected Results: All buttons are functional, have clear labels, and are appropriately sized.

4.1.6. Window Properties and Styling

Description: Verify the main window's overall size, the presence of a central widget, and custom styling.

Preconditions: `MainWindow` is initialized.

Steps:

1. Assert that the `MainWindow` has a reasonable size (e.g., width and height > 200 and < 2000).
2. Assert that a central widget exists and contains child components.
3. Assert that the `MainWindow` has a non-empty stylesheet and contains basic styling properties like 'background' or 'color'.

Expected Results: The window is well-proportioned, structurally sound, and visually styled.

4.1.7. Layout and Hierarchy

Description: Ensure that UI components are correctly parented and the overall layout is well-structured.

Preconditions: `MainWindow` is initialized.

Steps:

1. Iterate through all buttons and labels.
2. Assert that each component is a descendant of the `MainWindow`.

3. Assert that the central widget has a layout and that the layout contains child items.

Expected Results: All UI elements are correctly organized within the window hierarchy and layout.

4.1.8. Game State and Styling Consistency

Description: Verify the initial game state and consistent styling across game board cells.
Preconditions: `MainWindow` is initialized.

Steps:

1. Assert that the initial game mode displayed is 'VS AI' in test mode.
2. If at least two game cells exist, compare their stylesheets to ensure consistency.

Expected Results: The game starts in the expected mode, and game cells have uniform styling.

4.1.9. Memory Management

Description: Test the proper creation and destruction of `MainWindow` instances to prevent memory leaks.

Preconditions: None.

Steps:

1. Repeatedly create and destroy `MainWindow` instances within a loop.
2. Monitor for any memory-related issues or crashes.

Expected Results: Multiple `MainWindow` instances can be created and destroyed without memory leaks or crashes.

4.2. RegisterWindow Test Cases

The `RegisterWindow` handles user registration, including input validation and interaction with the database. Its tests focus on UI elements, input validation rules, password strength calculation, and registration logic.

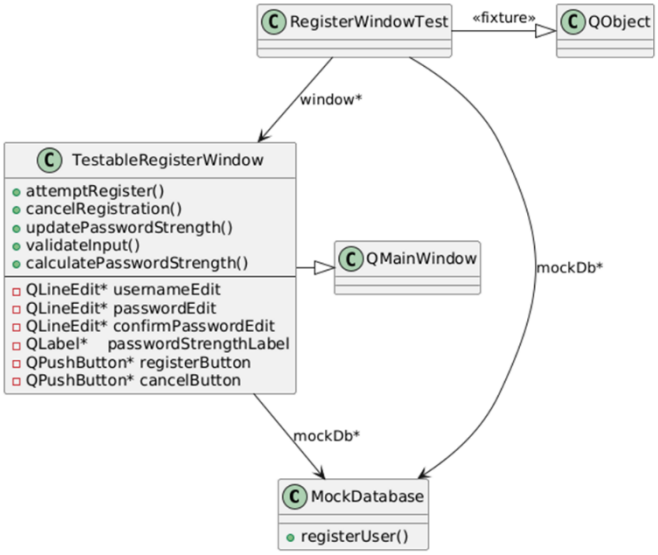

Figure 2--RegisterWindow test flow

4.2.1. UI Component Presence and Visibility

Description: Verify the presence of all necessary UI components and the correct echo mode for password fields.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Assert that username, password, confirm password `QLineEdit` s, password strength `QLabel` , and register/cancel `QPushButton` s are not null.
2. Assert that the password and confirm password fields have `QLineEdit::Password` echo mode.

Expected Results: All UI components are present, and password fields hide input.

4.2.2. Window Sizing

Description: Confirm that the `RegisterWindow` adheres to its minimum size constraints.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Assert that the window's minimum width is at least 350 pixels.
2. Assert that the window's minimum height is at least 450 pixels.

Expected Results: The window meets its minimum size requirements.

4.2.3. Input Validation - Empty Fields

Description: Test that the registration process prevents submission with empty username, password, or confirm password fields.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with various combinations of empty fields (e.g., empty username, empty password, all empty).
2. Assert that `validateInput` returns `false` for all empty field scenarios. Expected

Results: Input validation correctly identifies and rejects empty fields.

4.2.4. Input Validation - Username Length

Description: Verify that username input adheres to minimum (3 characters) and maximum (50 characters) length requirements.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with usernames shorter than 3 characters. 2. Call `validateInput` with usernames longer than 50 characters.
3. Call `validateInput` with usernames at the boundary lengths (3 and 50 characters).

Expected Results: `validateInput` returns `false` for invalid lengths and `true` for valid lengths.

4.2.5. Input Validation - Password Length

Description: Verify that password input adheres to minimum (6 characters) and maximum (100 characters) length requirements.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with passwords shorter than 6 characters. 2. Call `validateInput` with passwords longer than 100 characters.

3. Call `validateInput` with passwords at the boundary lengths (6 and 100 characters).

Expected Results: `validateInput` returns `false` for invalid lengths and `true` for valid lengths.

4.2.6. Input Validation - Username Special Characters

Description: Ensure that usernames containing special characters are rejected.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with usernames containing characters like '@', '.', ' ', '#'.
2. Call `validateInput` with usernames containing only alphanumeric characters.

Expected Results: `validateInput` returns `false` for usernames with special characters and `true` for alphanumeric-only usernames.

4.2.7. Input Validation - Password Mismatch

Description: Verify that registration fails if the password and confirm password fields do not match.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with matching password and confirm password.
2. Call `validateInput` with non-matching password and confirm password.

Expected Results: `validateInput` returns `true` for matching passwords and `false` for non-matching passwords.

4.2.8. Input Validation - Valid Input

Description: Confirm that valid combinations of username, password, and confirm password pass validation.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Call `validateInput` with several sets of valid credentials.

Expected Results: `validateInput` returns `true` for all valid input scenarios.

4.2.9. Password Strength Calculation

Description: Test the `calculatePasswordStrength` method for different password complexities.

Preconditions: `RegisterWindow` is initialized.
Steps:

1. Call `calculatePasswordStrength` with weak passwords (e.g., "123", "ab").
2. Call `calculatePasswordStrength` with medium passwords (e.g., "password", "Pass123").
3. Call `calculatePasswordStrength` with strong passwords (e.g., "Password1", "Password123!").

Expected Results: The method correctly returns "WEAK", "MEDIUM", or "STRONG" based on the password complexity.

4.2.10. Password Strength Label Update

Description: Verify that the password strength label dynamically updates its text and styling based on the entered password.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Simulate typing weak, medium, and strong passwords into the password field.
2. After each input, assert that the `passwordStrengthLabel` text contains the correct strength (WEAK, MEDIUM, STRONG) and its stylesheet reflects the corresponding color.

Expected Results: The password strength label updates in real-time with correct text and styling.

4.2.11. Button Functionality - Register and Cancel

Description: Confirm the existence and functionality of the Register and Cancel buttons.

Preconditions: `RegisterWindow` is initialized.

Steps:

1. Assert that the Register and Cancel buttons are not null and have the correct text.
2. Simulate a click on the Cancel button.

3. Assert that the `RegisterWindow` closes after the Cancel button is clicked.

Expected Results: Both buttons are present, correctly labeled, and the Cancel button successfully closes the window.

4.2.12. Registration Logic - Success and Failure

Description: Test the end-to-end registration process, including successful registration and handling of database failures.

Preconditions: `RegisterWindow` is initialized with a `MockDatabase` .

Steps:

1. Set `MockDatabase` to simulate successful registration. Input valid credentials and attempt registration. Assert that the `registrationSuccessful` signal is emitted and the `MockDatabase::registerUser` method is called once with correct data.
2. Set `MockDatabase` to simulate failed registration. Input valid credentials and attempt registration. Assert that the `registrationSuccessful` signal is NOT emitted and the `MockDatabase::registerUser` method is called once.

Expected Results: Successful registration emits the signal and updates the database; failed registration does not emit the signal.

4.2.13. Username Trimming Logic

Description: Verify that leading and trailing whitespace in the username is trimmed before registration.

Preconditions: `RegisterWindow` is initialized with a `MockDatabase` .

Steps:

1. Input a username with leading/trailing spaces (e.g., " trimmeduser ").
2. Attempt registration.
3. Assert that the MockDatabase::registerUser method receives the username without leading/trailing spaces (e.g., "trimmeduser").

Expected Results: Username is trimmed correctly before being sent to the database.

4.3. `Database` Test Cases

The `Database` class manages user authentication, registration, and game history. Its tests ensure data integrity, security, and correct retrieval. A dedicated test database (`test_tictactoe.db`) is used for isolation.

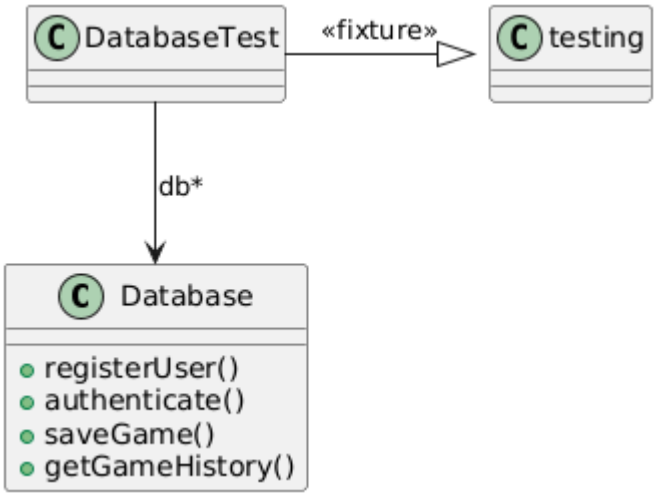

Figure 3-DataBase test flow

4.3.1. User Registration

Description: Verify the functionality of user registration, including valid and invalid scenarios.

Preconditions: A clean test database is set up.

Steps:

1. Attempt to register a user with valid credentials. Assert that registration returns `true`.
2. Attempt to register a user with a duplicate username. Assert that registration returns `false`.
3. Attempt to register a user with an empty username. Assert that registration returns `false`.

Expected Results: Valid users are registered, duplicate usernames are rejected, and empty usernames are rejected.

4.3.2. User Authentication

Description: Test the user authentication process for correct and incorrect credentials.

Preconditions: A clean test database is set up, and a user is registered.

Steps:

1. Authenticate with correct username and password. Assert that a valid user ID (greater than 0) is returned.
2. Authenticate with a valid username but an invalid password. Assert that -1 is returned.
3. Authenticate with a non-existent username. Assert that -1 is returned.
4. Authenticate with empty credentials. Assert that -1 is returned.

Expected Results: Only valid credentials result in successful authentication and a positive user ID.

4.3.3. Game Saving

Description: Verify the ability to save game states to the database.

Preconditions: A clean test database is set up, and a user is registered and authenticated.

Steps:

1. Save a valid game state for an authenticated user. Assert that saving returns `true`.
2. Attempt to save a game with an invalid user ID (e.g., -1). Assert that saving returns `false`.
3. Save multiple games for the same user. Assert that all saving operations return `true`.

Expected Results: Games are successfully saved for valid users, and invalid user IDs are rejected.

4.3.4. Game History Retrieval

Description: Test the retrieval of game history for various scenarios.

Preconditions: A clean test database is set up, and users are registered and authenticated.

Steps:

1. Retrieve game history for a user who has played no games. Assert that the message "No games played." is returned.
2. Save one or more games for a user. Retrieve game history and assert that it is not empty, contains expected board states (e.g., "XOX"), game results (e.g., "Win"), and timestamps (e.g., "Game at").
3. Retrieve game history for an invalid user ID. Assert that "No games played." is returned.
4. Save multiple games for a user and retrieve history. Assert that all saved games are present in the history and the count of game entries is correct.

Expected Results: Game history is accurately retrieved and formatted for valid users, and appropriate messages are returned for users with no games or invalid user IDs.

4.3.5. User Isolation

Description: Ensure that game data for one user does not interfere with or become accessible to another user.

Preconditions: A clean test database is set up.

Steps:

1. Register and authenticate two distinct users.
2. Save games for the first user.
3. Retrieve game history for both users. Assert that the first user has game history and the second user has "No games played."

Expected Results: User data is isolated, and game histories are specific to each user.

4.3.6. Full Workflow Integration

Description: An end-to-end test covering the complete lifecycle of user and game data interaction with the database.

Preconditions: A clean test database is set up.

Steps:

1. Register a new user.
2. Authenticate the user and obtain a user ID.
3. Save multiple games with different results (Win, Loss, Draw) for the

authenticated user.

4. Retrieve the complete game history for the user.

5. Assert that all saved game results (Win, Loss, Draw) are present in the retrieved history and the total count of game entries is correct.

Expected Results: The full workflow of registration, authentication, saving, and retrieving game data functions correctly and consistently.

4.4. `Game` Test Cases

The `Game` class encapsulates the core Tic-Tac-Toe game logic, including moves, win conditions, board state, and AI moves. Its tests ensure the correctness and fairness of the game.

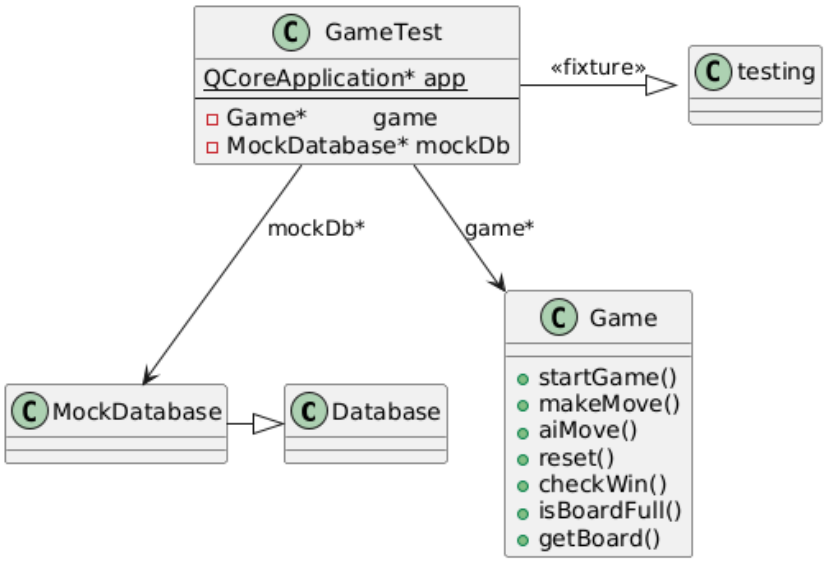

Figure 4-Game test flow

4.4.1. Constructor and Initialization

Description: Verify that the `Game` constructor correctly initializes the game board and AI flag.

Preconditions: None.

Steps:

1. Create a `Game` instance.
2. Assert that the `isVsAI()` flag is initially `false`.
3. Retrieve the game board and assert that all cells are empty

- Expected Results: The game board is empty, and the AI mode is off by default.

4.4.2. `startGame` Method

Description: Test that the `startGame` method correctly sets the AI flag and resets the game board.

Preconditions: `Game` instance is initialized.

Steps:

1. Call `startGame(true)`. Assert that `isVsAI()` returns `true`.
2. Call `startGame(false)`. Assert that `isVsAI()` returns `false`.
3. Make some moves on the board. Call `startGame(false)` again. Assert that the board is completely reset (all cells are empty).

Expected Results: The AI flag is set as expected, and the board is cleared upon starting a new game.

4.4.3. `makeMove` Method

Description: Verify the `makeMove` method's ability to place markers on valid positions and handle invalid scenarios.

Preconditions: `Game` instance is initialized.

Steps:

1. Make valid moves (e.g., (0,0), (1,1), (2,2)) for 'X' and 'O'. Assert that `makeMove` returns `true` and the board reflects the moves.
2. Attempt to make moves on out-of-bounds positions (e.g., (-1,0), (3,0)).

Assert that `makeMove` returns `false`.

3. Make a move on an occupied position. Assert that `makeMove` returns `false` and the original marker remains.

Expected Results: Moves are successfully made on valid, empty cells; invalid or occupied cells reject moves.

4.4.4. `checkWin` Method

Description: Comprehensive testing of all winning conditions (horizontal, vertical, diagonal) and scenarios with no winner.

Preconditions: `Game` instance is initialized.

Steps:

1. Horizontal Wins: For each row, set up a board state where 'X' or 'O' has three in a row horizontally. Assert `checkWin('X')` or `checkWin('O')` returns `true`.
2. Vertical Wins: For each column, set up a board state where 'X' or 'O' has three in a row vertically. Assert `checkWin('X')` or `checkWin('O')` returns `true`.
3. Diagonal Wins: Set up board states for both main and anti-diagonal wins. Assert `checkWin('X')` or `checkWin('O')` returns `true`.
4. No Win: Set up board states where no player has won (e.g., partial game, draw). Assert `checkWin('X')` and `checkWin('O')` return `false`.

Expected Results: The method accurately identifies all winning conditions and correctly reports no winner when applicable.

4.4.5. `isBoardFull` Method

Description: Verify the method that checks if the game board is completely filled.

Preconditions: `Game` instance is initialized.

Steps:

1. Assert `isBoardFull()` returns `false` on an empty board.
2. Make a few moves to partially fill the board. Assert `isBoardFull()` returns `false`.
3. Fill the entire board. Assert `isBoardFull()` returns `true`.

Expected Results: The method correctly indicates whether the board is full or not.

4.4.6. `reset` Method

Description: Ensure that the `reset` method clears the game board.

Preconditions: `Game` instance is initialized.

Steps:

1. Make several moves on the board.
2. Call `reset()` .
3. Retrieve the board and assert that all cells are empty.

Expected Results: The board is completely cleared after a reset.

4.4.7. AI Move Functionality

Description: Test the AI's ability to make strategic moves, including blocking opponent wins and making winning moves.

Preconditions: `Game` instance is initialized.

Steps:

1. AI Blocks Win: Set up a board state where the human player ('X') is one move away from winning. Call `aiMove('O')` . Assert that 'O' is placed in the blocking position.
2. AI Wins: Set up a board state where the AI ('O') is one move away from winning. Call `aiMove('O')` . Assert that 'O' is placed in the winning position and `checkWin('O')` returns `true` .
3. AI Makes Initial Move: Call `aiMove('X')` on an empty board. Assert that one 'X' is placed on the board.

Expected Results: The AI makes intelligent moves to block the opponent or secure a win, and can make a valid move on an empty board.

4.4.8. `getBoard` Method

Description: Verify that the `getBoard` method accurately copies the current state of the game board.

Preconditions: `Game` instance is initialized.

Steps:

1. Make several moves on the board.

2. Call `getBoard()` to retrieve a copy of the board.
3. Assert that the copied board matches the internal state of the `Game` object.

Expected Results: The `getBoard` method provides an accurate snapshot of the current game board.

4.4.9. Integration Test - Complete Game Scenario

Description: Simulate a full human-vs-human game from start to a winning condition.
Preconditions: `Game` instance is initialized.

Steps:

1. Call `startGame(false)` .
2. Make a sequence of moves for 'X' and 'O' that leads to 'X' winning (e.g., top row).
3. After each move, assert that `checkWin()` returns `false` until the winning move.
4. After the winning move, assert that `checkWin('X')` returns `true` .

Expected Results: The game progresses correctly, and the win condition is detected at the appropriate time.

4.4.10. Integration Test - Draw Game Scenario

Description: Simulate a full game that results in a draw.

Preconditions: `Game` instance is initialized.

Steps:

1. Make a sequence of moves for 'X' and 'O' that leads to a draw (board full, no winner).
2. After the last move, assert that `checkWin('X')` and `checkWin('O')` both return `false` .
3. Assert that `isBoardFull()` returns `true` .

Expected Results: The game correctly identifies a draw when the board is full and no player has won.

5. Conclusion

This testing documentation outlines the comprehensive testing efforts undertaken for the Tic-Tac-Toe application. By employing a combination of unit and integration tests, we have aimed to ensure the quality, reliability, and functionality of the application's core components, user management, and game logic. The detailed test cases provide a clear understanding of the tested functionalities and serve as a valuable resource for future maintenance and development.