Testing Documentation

Version: 1.0

Advanced Tic-Tac-Toe Game

Submitted by

Your Next Move Team

Team members:

Name	Section	ID
Mariam Ahmed Hamed	4	9220812
Mariam Mohamed Saeed	4	9220828
Malak Waleed Fouad	4	9220867
Menna Moutaz Elhosseiny	4	9220874
Maha Yasser Mohamed	4	9220878

Submitted to

Dr. Omar Nasr

Table of contents:

1. Purpose

This document outlines the strategies and results of testing efforts for the advanced Tic-Tac-Toe game, which includes user authentication, personalized game history, and an intelligent AI opponent. The testing ensures the reliability and correctness of the software through detailed test cases and coverage reports.

2. Testing Strategies: Unit Testing

- **Objective:** To verify that individual units of code (functions, classes, etc.) work as intended.
- **Tools Used:** QTest
- **Approach:**

Each unit test targets a small part of the application, typically a single function or class.

Mock objects and stubs are used to isolate the unit under test.

Test cases are designed to cover both typical and edge cases.

3. Detailed Tests and Cases

- `void testAIMove ()`

Initializes a game board and an AI player, sets up a specific board state and has the AI player make a move, and then checks to ensure that the board state has changed as a result of the AI's move. The test ensures that the `makeMove` function works correctly by validating that the board is updated.

Test case 1: The board is empty

Test case 2: The board is partially filled

- `void testBuildTree()`

Initializes a game board and an AI player, sets up an initial empty board state, calls the `build_tree` method to construct a decision tree, and then checks to ensure that the root node of the tree has children, which represent possible moves. The test ensures that the `build_tree` function works correctly by validating that the decision tree is populated with potential future moves.

- `void testMinimax()`

Test case 1:

- `board.setValue(0, 0, 0);` sets the board to an empty state.
- `std::numeric_limits<int>::min()`: Initial alpha value (negative infinity).
- `std::numeric_limits<int>::max()`: Initial beta value (positive infinity).
- `QVERIFY(score >= -1000);` checks that the score returned by the minimax function is at least -1000. This assertion ensures that the function behaves correctly for the given initial board state.

Test case 2:

- `board.setValue(0, 0, 1);` sets the position (0, 0) on the board to 1, indicating that the maximizer has made a move in that position.
- `aiPlayer.build_tree(root, -1);` constructs the game tree starting from the root node. The -1 indicates that the next player to move is the minimizer.
- `std::numeric_limits<int>::min()`: Initial alpha value (negative infinity).
- `std::numeric_limits<int>::max()`: Initial beta value (positive infinity).
- `false`: Indicates that the current player is the minimizer.
- `QVERIFY(score <= 1000);` checks that the score returned by the minimax function is at most 1000. This assertion ensures that the minimax function behaves correctly for the given board state.

Test case 3:

- `board.setValue(0, 0, -1);` sets the position (0, 0) on the board to -1, indicating that the minimizer has made a move in that position.
- `aiPlayer.build_tree(root, 1);` constructs the game tree starting from the root node. The 1 indicates that the next player to move is the maximizer.
- `std::numeric_limits<int>::min()`: Initial alpha value (negative infinity).
- `std::numeric_limits<int>::max()`: Initial beta value (positive infinity).
- `true`: Indicates that the current player is the maximizer.
- `QVERIFY(score >= -1000);` checks that the score returned by the minimax function is at least -1000. This assertion ensures that the function behaves correctly for the given board state.

Test case 4:

- `board.setValue(0, 0, 1);` sets the position (0, 0) on the board to 1, indicating that the maximizer has made a move in that position.
- `board.setValue(1, 1, -1);` sets the position (1, 1) on the board to -1, indicating that the minimizer has made a move in that position.
- `board.setValue(2, 2, 1);` sets the position (2, 2) on the board to 1, indicating that the maximizer has made another move in that position.
- `aiPlayer.build_tree(root, -1);` constructs the game tree starting from the root node. The -1 indicates that the next player to move is the minimizer.
- `std::numeric_limits<int>::min()`: Initial alpha value (negative infinity).
- `std::numeric_limits<int>::max()`: Initial beta value (positive infinity).
- `true`: Indicates that the current player is the maximizer.
- `5`: The depth limit for the minimax algorithm.
- `QVERIFY(score >= -1000);` checks that the score returned by the minimax function is at least -1000. This assertion ensures that the function behaves correctly for the given complex board state.

Test case 5:

- board.setValue(0, 0, -1); sets the position (0, 0) on the board to -1, indicating that the minimizer has made a move in that position.
- board.setValue(0, 1, 1); sets the position (0, 1) on the board to 1, indicating that the maximizer has made a move in that position.
- board.setValue(0, 2, -1); sets the position (0, 2) on the board to -1, indicating that the minimizer has made another move in that position.
- board.setValue(1, 0, 1); sets the position (1, 0) on the board to 1, indicating that the maximizer has made another move in that position.
- aiPlayer.build_tree(root, -1); constructs the game tree starting from the root node. The -1 indicates that the next player to move is the minimizer.
- std::numeric_limits<int>::min(): Initial alpha value (negative infinity).
- std::numeric_limits<int>::max(): Initial beta value (positive infinity).
- false: Indicates that the current player is the minimizer.
- QVERIFY(score <= 1000); checks that the score returned by the minimax function is at most 1000. This assertion ensures that the function behaves correctly for the given complex board state.

- void testEvaluate ()

Test cases 1-3:

- A game board is created with different scenarios of winning for the AI.
- The evaluate method of AIPlayer is called with this board.
- QCOMPARE(aiPlayer.evaluate(board2), 1000); this line asserts that the evaluation score is 1000, indicating a win for the AI.

Test cases 4-6:

- A game board is created with different scenarios of winning for the player.

- The evaluate method of `AIPlayer` is called with this board.
- `QCOMPARE(aiPlayer.evaluate(board6), -1000);` this line asserts that the evaluation score is -1000, indicating a win for the player.

Test case 7:

- A game board is created with a draw scenario.
- The evaluate method of `AIPlayer` is called with this board.
- `QCOMPARE(aiPlayer.evaluate(board7), 0);` this line asserts that the evaluation score is 0, indicating a draw.

- `void testCheckWin()`

Test cases 1-4:

Initializes a game board, sets up a specific board state where Player 1 wins by different scenarios, and then checks to ensure that the `checkWin` method correctly identifies Player 1 as the winner. The `checkWin` method is expected to return 1 if Player 1 has won. The `QCOMPARE` macro is used hereto assert that the result of `board.checkWin()` is equal to 1. The test ensures that the `checkWin` function works correctly by validating that it returns the correct result when Player 1 has won.

Test cases 5-8:

Initializes a game board, sets up a specific board state where Player 2 wins by different scenarios, and then checks to ensure that the `checkWin` method correctly identifies Player 2 as the winner. The `checkWin` method is expected to return -1 if Player 2 has won. The `QCOMPARE` macro is used hereto assert that the result of `board.checkWin()` is equal to -1. The test ensures that the `checkWin` function works correctly by validating that it returns the correct result when Player 2 has won.

Test case 9:

`QCOMPARE(board.checkWin(), 2);` this line calls the `checkWin` method of the `GameBoard` class to determine if there is a winning condition on the board. In this case, `checkWin` is expected to return 2 to indicate a draw. The

QCOMPARE macro is used here to assert that the result of board.checkWin() is equal to 2. This function initializes a game board, sets up a specific board state where the game ends in a draw (all positions filled with no winner), and then checks to ensure that the checkWin method correctly identifies the draw condition. The test ensures that the checkWin function works correctly by validating that it returns the correct result when the game ends in a draw.

- void testGameNotOver()

Initializes a game board, sets up a specific board state where both players have made moves but no winner has been determined, and then checks to ensure that the checkWin method correctly identifies the game as not over. The test ensures that the checkWin function works correctly by validating that it returns the correct result when the game is ongoing and no winner has emerged yet.

- void testInitialEmptyBoard()

Initializes a game board and checks that its initial state is correctly recognized as empty by the checkWin method. This test ensures that the checkWin function works correctly by validating its behavior when no moves have been made on the board.

- void testSetGetValue()
- board.setValue(0, 0, 1); sets the value at position (0, 0) on the board to 1.
- board.setValue(1, 1, -1); sets the value at position (1, 1) on the board to -1.
- board.setValue(2, 2, 0); sets the value at position (2, 2) on the board to 0.

- QCOMPARE(board.getValue(0, 0), 1); asserts that the value retrieved from position (0, 0) on the board using the getValue method is 1.
- QCOMPARE(board.getValue(1, 1), -1); asserts that the value retrieved from position (1, 1) on the board is -1.
- QCOMPARE(board.getValue(2, 2), 0); asserts that the value retrieved from position (2, 2) on the board is 0.

Modify the values and check again

- board.setValue(0, 0, -1); changes the value at position (0, 0) on the board to -1.
- board.setValue(1, 1, 1); changes the value at position (1, 1) on the board to 1.
- board.setValue(2, 2, 1); changes the value at position (2, 2) on the board to 1.
- QCOMPARE(board.getValue(0, 0), -1); asserts that the value at position (0, 0) on the board is now -1.
- QCOMPARE(board.getValue(1, 1), 1); asserts that the value at position (1, 1) on the board is now 1.
- QCOMPARE(board.getValue(2, 2), 1); asserts that the value at position (2, 2) on the board is now 1.

These assertions ensure that the setValue and getValue methods of the GameBoard class correctly set and retrieve values at specific positions on the board, both before and after modifications.

- void testOutOfBoundsAccess()

- QCOMPARE(board.getValue(3, 0), 0); attempts to retrieve the value from position (3, 0) on the board using the getValue method. Since this position is out of the board's bounds, the expected result is 0, which is the default value returned by the getValue method for out-of-bounds accesses.
- QCOMPARE(board.getValue(0, 3), 0); attempts to retrieve the value from position (0, 3) on the board. Similarly, this position is out of the board's bounds, so the expected result is 0.

These assertions verify that the `getValue` method correctly handles out-of-bound accesses by returning a default value (0 in this case) without causing any errors or exceptions.

- `void testLogin()`
  - Opens an in-memory SQLite database (memory). This database only exists during the execution of the test and is useful for unit testing because it does not persist any data.
  - Defines a SQL statement to create a players table with columns id, email, password, and `last_login_date`.
  - Defines a SQL statement to insert a test user with email test@example.com and password `testpassword` then executes the SQL statement and checks if it was successful.
  - Attempts to log in using the correct email and password then checks if the result of the login method is the same as the email used for login, indicating a successful login.
  - Attempts to log in using the correct email but the wrong password checks if the result of the login method is an empty string, indicating a failed login due to incorrect password.
  - Attempts to log in using an email that does not exist in the database checks if the result of the login method is an empty string, indicating a failed login due to non-existent user.

- `void testSignup()`
  - Opens an in-memory SQLite database (memory), which is useful for unit testing because it does not persist any data beyond the duration of the test.
  - Defines a SQL statement to create a players table with columns id, email, password, name, age, city, and `current_date` then executes the SQL statement and checks if it was successful.
  - Attempts to sign up a new user with the given details (email, password, name, age, city) then checks if the result of the signup method is the same as the email used for signup, indicating a successful signup.

- Attempts to sign up again with the same email then checks if the result of the signup method is an empty string, indicating a failed signup due to the email already existing in the database.
- Defines a SQL statement to select the user details from the players table where the email matches test@example.com and verifies that the data returned from the query matches the expected values (email, password, name, age, city).

- `void testCustomHash()`

- Defines a string `str` with the value "`teststring`".
- Hashes the string twice, storing the results in hash1 and hash2.
- Asserts that hashing the same string twice produces the same hash value, ensuring the hash function is deterministic.
- Defines a different string `differentStr` with the value "`differentstring`".
- Hashes this different string, storing the result in `differentHash`.
- Asserts that hashing different strings produces different hash values, ensuring the hash function can distinguish between different inputs.

- `void testHashPassword()`

- Defines a password `password` with the value "password123".
- Hashes the password, storing the result in `hashedPassword`.
- Asserts that the hashed password is not empty, ensuring the hash function produces a valid output.
- Hashes the same password again, storing the result in hashedPassword2.
- Asserts that hashing the same password twice produces the same result, ensuring the hash function is deterministic.
- Defines a different password `differentPassword` with the value "`differentpassword`".Hashes this different password, storing the result in `differentHashedPassword`.
- Asserts that hashing different passwords produces different hash values, ensuring the hash function can distinguish between different inputs.