Test Plan

This test suite for Camel Caravan Project has 75+ test cases:

1. Automated Testing with OUnit:

The majority of the system is tested automatically using OUnit and QCheck.

- Automated tests include:
- Conversion of ship states to strings (empty, number, one, incorrect, two).
- Board creation and its dimensions.
- Validity checks for player guesses.
- Updates to the game grid based on player guesses.
- Placement of camels on the board.
- Checking if camels can be placed at specific spots.
- Ensuring unique spots for camel placement.
- Generating and verifying proximity hints.
- Reading and parsing user input for guesses and hints.
- Player-specific game state updates (e.g., recording guesses, marking camels as found).

2. Manual Testing:

Manual testing was performed to ensure the integration of various components, particularly:

- User interactions and UI elements.
- Network communication between client and server (connection, message handling).
- End-to-end gameplay scenarios.

3. Modules Tested by OUnit:

- "Battleship.Board" and "Battleship.Ship" modules were primarily tested using OUnit.
- Test cases were developed using a combination of:
- Black Box Testing: Focused on the functionality described in the specifications without looking at the code implementation. Example: Testing the "to_string" function for different camel states.
- Glass Box Testing: Developed by understanding the internal code structure. Example: Testing "can_place_camel" to ensure proper handling of edge cases.
- Randomized Testing: Using QCheck to generate random scenarios for camel placement and verifying the placement logic.

4. Argument for Testing Approach:

- The combination of automated and manual testing ensures comprehensive coverage of the system.
- Automated tests verify the correctness of core functionalities and edge cases while manual tests complement automated tests by covering interaction and integration aspects.
- The use of black box, glass box, and randomized testing strategies ensures that the tests cover a wide range of scenarios, like edge cases and random inputs.
- The QCheck properties provide an additional layer of assurance by running a large number of tests with randomly generated inputs, increasing the likelihood of catching unforeseen issues.

To Note: Since our game is more heavy on the main board interface and randomization, we faced some challenges with testing and figuring out what to test in the first place. For example, in our main.ml we have a lot of printing statements that cannot be tested. However, after using QCheck and adding more functionality to Board.ml, we were able to cover a wider range of test cases. Adding on, just through playing the game many times we were also able to identify areas of improvement that we then reimplemented and tested again. Overall, we believe that our testing approach demonstrates the correctness of the system through combining automated tests with thorough manual testing, covering both unit-level and integration-level aspects of the game.