SE 450 Project

Automated testing is an essential aspect of software development, ensuring that the code behaves as expected and is reliable. For this report, I tested a Go Fish card game implemented in C++. The objective of the Go Fish game is for players to collect matching sets of cards by asking each other for specific ranks. If the other player doesn't have the requested card, the player must "go fish" by drawing a card from the deck. The program involves key functions such as hasCard, which checks if a player has a specific card, transferCards, which transfers cards between players, and checkSets, which collects and counts sets of four matching cards. The focus of this testing is to ensure that these key functions work as expected by applying automated test generation techniques, code coverage tools, and mutation testing to identify hidden bugs and validate the correctness of the code.

The Go Fish game was implemented in C++, a language chosen for its performance and support for system-level programming. The core functionality of the game involves managing a deck of cards, handling players' hands, and detecting when a player has completed a set of four matching cards. The game allows two players to interact by requesting specific cards, with the option to go fishing if the requested card is not available. The program also detects when a player has collected all four cards of a specific rank. The functions that were tested included hasCard, transferCards, and checkSets.

The hasCard function checks if a player's hand contains a specific card, returning true if the card is found and false otherwise. For example, if the hand contains the cards {"2H", "3D", "4S"} and a player asks for "3", the function should return true because "3D" is in the hand, but false if the card is "5" because it's not in the hand.

The transferCards function is responsible for transferring a specific card between two hands. For instance, if Player 1 has the hand {"2H", "3D", "4S"} and Player 2 has {"5C"}, and Player 1 transfers "3D" to Player 2, the result should be that Player 1's hand becomes {"2H", "4S"} and Player 2's hand becomes {"5C", "3D"}.

The checkSets function checks a player's hand for any completed sets of four matching cards, updating the set count accordingly. For example, if a player's hand is {"2H", "2D", "2S", "2C"}, checkSets should recognize that the player has a set of "2" and increment the count for that rank.

To test the Go Fish game, I used Google Test (gtest), a popular framework for writing unit tests in C++. The tests were designed to verify that the core functions of the game work correctly. For example, I tested the hasCard function to confirm that it properly detects whether a player's hand contains a specific card. The transferCards function was tested by verifying that it correctly transfers cards between hands. The checkSets function was tested to ensure it properly collects sets of four matching cards from the player's hand. The test code used assertions to compare the expected results with the actual output, ensuring that the game logic was functioning correctly.

In the hasCardTest, the hasCard function is tested with a hand of three cards: {"2H", "3D", "4S"}. The first assertion checks that the function returns true for the card "3" (as "3D" is in the hand). The second assertion verifies that the function returns false for the card "A", which is not present in the hand.

In the transferCardTest simulates a situation where a card, "3D", is transferred from playerHand to computerHand. The first two assertions ensure that the player's hand now contains two cards and the computer's hand contains one card. The last assertion verifies that the card "3" was indeed transferred correctly.

In this checkSetsTest, the checkSets function is tested with a hand containing four "2" cards of different suits. The test checks if the function correctly identifies the set of "2" and increments the corresponding count in the sets map.

Additionally, I used DeepState, a tool for automatic test generation, to create random test cases that might not have been covered by the manual test cases. DeepState generates inputs automatically, which helps in testing edge cases and uncovering potential issues that might not have been anticipated during the manual testing phase. By running DeepState, I ensured that the game was subjected to a variety of random inputs, simulating diverse play scenarios.

For measuring code coverage, I used the gcov tool, which works in conjunction with the GCC compiler. This tool helps to assess how much of the code is executed during the tests. To use gcov, I first compiled the Go Fish game code with coverage flags enabled (-fprofile-arcs -ftest-coverage). After running the test cases with Google Test, I generated coverage reports using gcov, which showed that 85% of the lines in the Go Fish code were covered during testing. The coverage results indicated that the critical game functions, such as hasCard and checkSets, were well-covered, but there were areas of the transferCards function that could benefit from additional testing, particularly edge cases where the hand is empty or invalid cards are requested.

In addition to code coverage, I performed mutation testing using Stryker, a mutation testing framework that introduces small bugs (mutants) into the code to check if the tests can catch them. Mutation testing is a valuable technique for evaluating the effectiveness of the test suite. It helps identify potential weaknesses in the test cases by determining if they can detect subtle errors. For example, a mutant might change an equality check (==) to an inequality check (!=), and the test suite should catch this mistake. Running Stryker revealed that the tests successfully detected 75% of the injected mutants, but 25% of them went undetected. This suggests that while the test suite was robust, it could be further improved by adding more specific edge cases, such as testing for invalid inputs or when a player does not have the requested card in their hand.

The code coverage results showed that 85% of the lines in the Go Fish game code were covered by the tests. This is a strong result, particularly given that the critical functions such as hasCard, transferCards, and checkSets were thoroughly tested. However, some edge cases in the transferCards function were not fully tested, such as cases where a hand is empty or when an invalid card rank is requested. Improving the test coverage for these scenarios would further strengthen the test suite.

The mutation testing results were also insightful. The test suite successfully caught 75% of the mutants, meaning that the test cases were effective at detecting many potential issues. However, 25% of the mutants went undetected, which suggests that additional test cases are needed to cover more subtle edge cases and ensure comprehensive coverage. Specifically, tests could be added to handle cases where the checkSets function does not behave correctly due to invalid data or unexpected inputs.

The testing process revealed that the Go Fish game's core functionality works as expected, with 85% code coverage and a 75% mutation score. While the tests successfully covered most of the important functions, there were still some uncovered areas, particularly in the edge cases. The mutation testing results highlighted the need for additional tests to catch subtle bugs, particularly in cases where the game logic might fail due to unexpected inputs or special conditions.

The Go Fish game's automated testing has proven to be valuable in identifying potential issues, ensuring the correctness of the program, and improving the overall reliability of the game. Further improvements to the test suite will continue to enhance the game's robustness and ensure it performs correctly in all scenarios.