CSE 565: Software Verification and Validation MCS Project Portfolio Report

1st Andy Gonzalez
School of Computing and Augmented Intelligence
Arizona State University
Tempe, Arizona
agonz600@asu.edu

***Abstract*—This report summarizes the four projects developed in the course CSE 579: Knowledge Representation and Reasoning. In this report, we go over each of the four projects: Software Unit Testing Framework, Specification-based Testing, Design of Experiments, and Structural-Based Testing projects. We give an brief introduction to each project, a description of the solution, a description of the results, a statement of contributions, and a self-reflection.**

***Index Terms*—Design of Experiments, Unit-testing, Structural-testing, Specification-testing, Verification, Validation**

I. Introduction

These projects were developed in order to showcase different verification and validation testing methods and principles. We cover four different testing methods.

A. Software Unit Testing Framework Project

a) Description of Method: A unit test is an automated test that quickly verifies a small unit of code in an isolated manner [1, pp. 21]. We can isolate a unit of code by using techniques such as mocking, creating mock objects to simulate real objects, or stubbing, creating stub objects that simulate the object just enough for the code to run. The purpose of unit-testing is to enable the continual growth of a software project [1, pp. 6]. This implies that unit-testing needs to be done concurrently with development in order to ensure the successful delivery of software.

b) Problem Description: The first task of this project was to create a heap sort algorithm using any resources available. The second task was to find a unit testing framework for a programming language of choice. The third task was to ask a generative AI tool to generate test cases using the chosen framework. The last task was to asses the test cases and improve upon them.

B. Specification-based Testing Project

a) Description of Method: Specification-based testing is creating test cases based on the specifications of the program as a point of reference [6]. The specifications of a program are a formal document that state the input and output requirements. Specification-based testing is a broad testing paradigm. In this project we used the specification-based testing techniques: equivalence class partitioning, boundary-value analysis, and cause-effect analysis.

Equivalence class partitioning (EP) involves the grouping of similar inputs or similar outputs, each group is an equivalence class. The idea is that the function being tested either works for a single input in the equivalence class or works for none at all [7, pp. 1559].

Boundary value analysis (BV) involves testing the elements at the boundaries of equivalence classes. The intuition behind this is that programs that work for values inside equivalence classes tend to fail on values on the edges of equivalence classes [7, pp. 1560].

Cause effect analysis is the use of decision tables or decision trees to generate test cases. In decision tables, the inputs are rows and the columns are test cases. Each row is an equivalence partition of an input or output. This techniques involves testing multiple valid partitions per test or one invalid partition per test. Decision trees are a graphical representation of decision tables where inputs are nodes and equivalence partitions are edges. Decision tables and decision trees do not necessarily produce the same test cases.

b) Problem: The problem we aimed to solve in this project was finding ten seeded defects by testing a product discount calculator application given a requirements sheet using specification-based testing techniques. The inputs for the program are User Name, Age, User Status, Rewards Member Status, Season Bought, Product Category, and Rating of Product. Each input has its requirements such as data type and domain of its valid values. The outputs are error messages for incorrect type of Username, Age or Product, no discount if product category is unknown, no discount for new users, four combinations with four possible discounts, and no discounts for any other combination.

C. Design of Experiments Project

a) Description of Method: In this project, we used a Design of Experiments (DOE) testing technique pairwise-combination testing. Pairwise combination testing is a form of t-way combination testing, specifically when $t = 2$. This means that we aim to test every possible pair of n inputs. The intuition behind this method of testing is known as the interaction rule, which states that most failures are caused by a single parameter, or interactions between two-to-six parameters [8, pp. 19]

b) Problem: The goal of this project was to generate test cases to test a mobile application based on given specifications. We needed to ensure that the app worked on different possible combinations of types of phone, authentication, connectivity, memory, and battery level. We needed to use a pairwise combination tool, use a generative AI tool, deliver two sets of test cases, and compare the test cases.

D. Structural Testing Project

a) Description of Method: In this project we focused on two structural-based testing techniques: code coverage and static analysis. Code coverage is the percentage of executable elements in a program that were executed [9, pp. 51]. Static analysis is an examination of design, requirements, and code without executing the program. The goal of static analysis is to find defects [9, pp. 72].

b) Problem: We were given two Java files in this project, $VendingMachine.java$ and $StaticAnalysis.java$.

For the first part of this project, we needed to write unit-tests for the $VendingMachine.java$ achieving 100% statement coverage and at least 90% branch coverage. We were given a set of requirements: input is an integer and one of three possible product names, it returns the product and change, and it returns the change missing and other products they can buy if there is not enough money. To assess code coverage, we needed to find a code coverage tool compatible with Java.

For the second part, the goal was to find two seeded data-flow anomalies in $StaticAnalysis.java$. To do this, we needed to research and use a static analysis tool compatible with Java.

II. Solution

A. Software Unit Testing Framework Project

In this project, we used Java 21 and JUnit 5. JUnit 5 is a unit testing framework that provides a comprehensive set of features such as dependency injection, it integrates with easily integrates with build tools and IDEs [5]. The generative AI tool we used was ChatGPT 3.4. ChatGPT is a popular generate AI chat-bot developed by OpenAI that has answers natural language queries in a conversational manner [4]. To manage the project and its dependencies we used the build tool Maven.

a) Development of the Heap-Sort Algorithm: Heap-Sort is a very common algorithm to sort arrays. The intuition behind the algorithm is to use the properties of a Max-heap data structure. A max-heap data structure is a binary tree data structure in which each internal node's value is greater than the values of its children nodes [2]. In heap-sort we convert the array into a max-heap initially and define a variable i equal to the length of the array. Then we complete the steps until $i = 0$.

1) Swap the top element with element at index i
2) Convert the sub-array, from index 0 to index i, into a max-heap.
3) Decrease i

The exact code used in this project can be found under the Java implementation at [3] with a time complexity of $O(n \log n)$. This is an in-place sorting algorithm, meaning no additional data-structure are used during the algorithm.

b) Generation of Test Cases: The test cases were generated by asking ChatGPT the following prompt: "Generate unit-level test cases for heap-sort algorithm using JUnit." The AI model generated seven unit test cases:

- Empty array
- Single element array
- Already-sorted array
- Reverse-sorted array
- Unsorted array
- Array with duplicates
- Array with negative numbers

The test cases were given in functional Java code using JUnit appropriately. We copied the code into the test file and ran the tests.

B. Specification-based Testing Project

To find all the seed defects, we first divided the inputs into equivalence classes.

- Username. Usernames with less than five characters, five to ten characters, greater than ten characters, with number, with hyphen, with unmentioned character, or with underscore.
- Age. Age less than 18, greater than or equal to 18, or not an integer.
- Rating of product. Ratings less than 1, greater than 10, between 1 and 10, and non-integer.

The other inputs were stated in the requirements document as not needing to be verified. Once we identified the equivalence classes, we also identified the boundary values at the edge of the equivalence classes. For example, username's boundary values were 4, 5, 10, and 11.

Next, we created a decision table to find the number of test cases. In our decision table, we included each possible output as a row. To minimize the number of tests, we used valid boundary values to as input values to test the valid test cases. The decision table yielded 18 test cases, 10 invalid test cases, and 8 valid test cases.

C. Design of Experiments Project

The testing tool used in this project was AllPairs, specifically allpairs4j for Java. This tool supports t-way test combination generation and supports dependent inputs through use of constraints. The generate AI tool used was Microsoft's Copilot using GPT-4. The building tool utilized was Maven.

a) Generating Test Cases using AllPairs: The test cases were very simple to generate using allpairs4j. First, we added the necessary dependencies to project. Then, we used the $allPairsBuilder$ function entering parameter names and possible values, and run the project. Lastly, the tests were printed to standard output by compiling and running the project.

b) Generating Test Cases using Copilot: To generate the test cases, the following prompt was asked after entering in all parameters and their values, "Can you generate a complete set of pairwise combinations test cases for the five above parameters? Note a test case consist of the five parameters and each test can cover multiple pairs at once." The response yielded the test cases.

D. Structural Testing Project

The Java code coverage tool used to complete part 1 was JaCoCo. The static analysis tool used to complete part 2 was PMD. For both parts, we copied the given files into separated Maven projects, added the necessary plugins, and configured the plugin to run on successful compilation.

a) Part 1: Code Coverage Solution: To find code coverage, we need to create test cases according to the input specifications. To do this, we copied the specifications into ChatGPT and asked it to generate a set of test cases that achieve complete code coverage according to the specifications. It generated a total of 7 test cases covering the following: buying one of three products with the exact change (3 test cases), buying one of three products with insufficient change (3 test cases), and buying a product with too much change (1 test case).

For the results, we compiled the project and a code coverage report was generated automatically in HTML.

b) Part 2: Static Analysis Solution: After the necessary plugin and configurations were added to the build file, we compiled the project to automatically get an HTML report of the static analysis of the file.

III. Results

A. Software Unit Testing Framework Project

The seven generated test cases were ran and the seven test cases passed.

a) Improvement of Test Cases: In order to improve the test cases, we brainstormed a few other scenarios that were not covered by the generated cases. Here were the brainstormed test cases:

- Array with two integers
- Large array
- Array with large numbers

The idea behind the first test case is to test the case smallest possible case in which the program enters the recursive loop. An array with a single element is a base case of the heap-sort algorithm and does not enter the loop. The other two test cases are fairly standard test cases that should be tested.

Once the code for these new test cases was added, the code was ran again. All ten cases were successful.

b) Assessment of AI tool and Unit Testing Framework: JUnit integrated automatically with the creation of a Maven project. The code stubs were created automatically as part of the creation of the project. The test cases were copy-and-pasted and worked without any errors.

To generate the unit test cases for the heap-sort algorithm, ChatGPT performed reasonably well. It covered most of the most important test cases developers need to be concerned about such as the edge cases of an empty array and an array with a single element. Furthermore, the AI tool required a simple prompt to create a comprehensive suite of test cases. The results showcase the effectiveness of modern AI tools in assisting developers with the creation of test cases by delivering functional and context-based code.

B. Specification-based Testing Project

a) Test Cases Results: Out of the 18 test cases, 11 failed. In more detail, 6 invalid test cases were ran against the program and ran successfully, 5 valid test cases were ran against the program and failed to run successfully. Note that invalid test cases are meant to fail, and valid test cases are meant to pass. By inspecting and comparing the expected and the actual output, ten defects were found.

b) Defects found: Here is a summary of the defects found

- New Users receive a ten percent discount
- A specific set of users receive a 5% discount when they should have received 15%. Two more similar defects.
- Username of 11 characters is permitted.
- Username with underscore is permitted.
- User with Age of 17 is permitted.
- Rating of 0 is permitted.
- Non-integer rating is not handled.
- Non-integer user age is not handled.

By successfully finding the ten seeded defects, we met our testing goals and we showcased the efficacy of specification-based techniques in finding defects.

C. Design of Experiments Project

a) AllPairs result and analysis: The testing tool generated 28 test cases. Upon manual verification, these test case covered all possible pairs and we achieved our testing goal of complete pairwise combination coverage.

AllPairs proved to be an effective and easy-to-use tool in pairwise combination testing.

b) Copilot result and analysis: Microsoft's Copilot generated only 20 test cases. Upon inspection, there were missing pairs that needed to be tested. After prompting it to write more test cases, it would generate duplicated test cases. Therefore, we failed to meet our testing goals with the use of this tool.

Copilot proved to be lackluster in generating pairwise combinations. The generated AI model seemed to be hesitant in doing computation. Initially, it would only generate a few test cases and refuse to generate more. It took a few attempts to get the prompt wording precisely correct for Copilot, so that it would produce the expected results. Other generative AI tools also proved lackluster. At first, we attempted to use ChatGPT 3.4. When prompted, it responded with trying to write a program to generate pairwise combinations, repeatedly failed, and stopped computations as its computation limit was reached for the day. We can infer that the hesitancy of generative AI conversational models to generate pairwise combinations is due to combinatorial nature of the problem. The models most likely assumed that the number of test cases would be too large to calculate, thus refusing to do so.

D. Structural Testing Project

a) Code Coverage Results: The code coverage reported by JaCoCo for the initial set of test cases was 95% statement coverage and 93% branch coverage. We achieved our branch coverage goal and surpassed it by 3%, however we still needed to achieve complete statement coverage. Upon clicking through the project name in the JaCoCo report, and then the file name, we saw that the line missing coverage was the class declaration. To achieve 100% statement coverage, we added a test case that created an object of the class and tested the object. We re-compiled the project, analyzed the results once again, and achieved 100% statement coverage. We met our testing goals.

b) Static Analysis Results: After running PMD and generating a report, we saw that two anomalies of the same type were detected. There were to unused local variables called "weight" and "length". This met our testing goals of finding two seeded defects.

To assess the quality of the static analysis tool, we inspected the code and found a missed anomaly that was typically detect by static analysis tools but was not detected by PMD. The anomaly was the use of the equality operator to compare strings. This is an anomaly because the equality operator in Java when used to compare strings, compares the references of string objects. This causes errors if the function is ever used with allocated string objects.

IV. Contributions

Each project describe in this report was developed individually by me. For each project, I was aided by generative AI tools such as ChatGPT and Copilot in tasks such as the generation of test cases, which was required and allowed by the project requirements.

V. Reflection

In completing the first project, I learned the fundamentals of unit testing. I learned how to create test cases that verify the correctness of the intended functionality.

In completing the second project, I learned how to test code based on given specifications. I learned how to divide the input and output into equivalence classes, how to define the boundary values, and develop test cases using these with decision tables and trees.

In completing the third project, I learned how to efficiently generate all 2-way combinations for the design of experiments method pairwise combination testing.

In the last project, I learned how to use code coverage tools and static analysis tools, and interpret their reports.

Through the completion of these four projects, I learned many verification's and validations techniques to adequately test programs. I learned how to integrate a variety of testing tools in the workflows of developers and I became comfortable with reading through the documentation of these tools. Lastly, I learned the strengths and weaknesses of popular generative AI tools in aiding testers during test development, and in which situations they can be leveraged to hasten the workflow.

References

[1] Vladimir Khorikov, Unit Testing Principles, Practices, and Patterns. Simon and Schuster, 2020.

[2] "Introduction to Max-Heap – Data Structure and Algorithm Tutorials," GeeksforGeeks, Mar. 23, 2023. https://www.geeksforgeeks.org/introduction-to-max-heap-data-structure/ (accessed Jul. 14, 2024).

[3] "Heap Sort - Data Structures and Algorithms Tutorials," GeeksforGeeks, Mar. 16, 2013. https://www.geeksforgeeks.org/heap-sort/#

[4] OpenAI, "OpenAI," OpenAI, 2023. https://openai.com

[5] S. Bechtold et al., "JUnit 5 User Guide," junit.org. https://junit.org/junit5/docs/current/user-guide/

[6] "Specification-Based Testing," GeeksforGeeks, Oct. 20, 2022. https://www.geeksforgeeks.org/specification-based-testing/

[7] A. Bhat and S. Quadri, "Equivalence Class Partitioning and Boundary Value Analysis - A Review," 2nd International Conference on Computing for Sustainable Global Development, 2015.

[8] D. Richard Kuhn, R. N. Kacker, and Y. Lei, "Combinatorial coverage as an aspect of test quality," vol. 28, no. 2, pp. 19–23, Mar. 2015.

[9] D. Graham, E. Van Veenendaal, I. Evans, and R. Black, "FOUNDATIONS OF SOFTWARE TESTING ISTQB CERTIFICATION." Accessed: Jul. 14, 2024. [Online]