Software Verification and Validation Project Report

Ari Argoud
School of Computing and Augmented Intelligence
Ira A. Fulton Schools of Engineering
Arizon State University
Tempe, Arizona, United States
aargoud@asu.edu

***Abstract*—The following portfolio report details three projects completed for CSE565: Software Verification and Validation. Parts one and two of the first project are considered separately for organizational purposes.**

In the first project, two parts demarcate the exploration of specification-based testing methodologies. Part one introduces Equivalence Partitioning, Boundary Value Analysis, and Cause and Effect Testing in the context of black-box software. Part two extends this inquiry through the implementation of Design of Experiments (DOE) in the context of a hypothetical mobile application. The second project focuses on structural-based testing, dividing its attention between code coverage and static source code analysis. Utilizing the IntelliJ Code Coverage Runner and PMD plugin respectively, a Vending Machine application serves as the medium to study statement and branch coverage, while a shipping cost calculation application provides a framework for exploring static analysis. The third and final project examines GUI testing in the context of software verification. Through the architecture and testing of two slightly different versions of a simple web application, the GUI test automation tool Selenium is assessed for facility and functionality with an emphasis on test reusability. Together, these projects furnish a generalized perspective on software verification and validation and provide insight for the integration of several tools into an effective testing arsenal.

I. Project 1 Part 1: Specification-Based Testing

A. Introduction

Specification-based testing, or black-box testing, allows testers to assess the functionality of an application through only its inputs and outputs, without the need to understand implementation. In this first project, a test matrix is constructed based on a given application and given lists of input and output requirements. This is achieved through a combination of equivalence partitioning, boundary value analysis[1], and cause and effect testing.

Equivalence partitioning refers to the technique wherein the range of inputs is divided into equivalence classes to minimize quantity of required tests.

Boundary value analysis further specifies that only valid values directly on the boundary of a requirement, or invalid values directly outside the boundary of a requirement should be tested. This creates a more efficient testing strategy for error discovery. Naturally boundary value analysis is only applicable to requirements measuring continuous attributes, ordered set attributes, and the like.

Cause and effect testing enumerates combinations of dependent inputs to trace errors in dependency logic and thereby correctly satisfy output requirements.

To construct the matrix, test cases are recorded with a `Test Case Number`, and their specified input variables `Name, Age, User Status, Rewards Member Status, Season Bought, Product Category`, and `Rating`. This is followed by a column denoting whether the test behaves as expected `(Pass/Fail)`, and a column mapping each case to its requirement(s) `(Requirements Mapping)`, as specified in the provided overview document. Cases are run in aggregate via bash script.

B. Solution

To assess the input requirements and independent output requirements, equivalence partitioning is coupled with boundary value analysis to guide and limit the quantity of tests created where applicable. Cause and effect testing is then utilized to assess the output requirements on the various dependent inputs.

The `Name` attribute was tested with strings of length 4, 5, 10, and 11 to fulfill the boundaries of the requirement that Names can only be 5-10 characters long. Names containing numbers, hyphens, and underscores were added to the matrix to test the requirements disallowing names containing these characters. Names containing each possible character were created to determine if the program was robust to unexpected characters.

The `Age` and `Rating` attributes were tested similarly. Each had a singular input requirement; only ages over 18 and ratings between 1-10 were allowed. Test cases were created to assess these boundaries. Additionally, these variables were also tested at and beyond the border of integer overflow and underflow, to further assess robustness to unexpected inputs.

To conclude the input requirement testing, valid cases were overlapped where possible to minimize the testing space. Valid cases were generated to ensure each categorical attribute was tested with each of their values at least once.

To test the output requirements, error messages generated from incorrect usernames, ages, and ratings were assessed. Following that, cause and effect analysis was performed by enumerating the combinations of `User Status, Rewards Member Status, Season Bought`, and `Product Category`, and pairing each of these cases with valid `Name, Age`, and `Rating`.

C. Results

What follows is an accounting of all the defects found during testing. The following three combinations of attributes resulted in incorrect discounts: returning customers with gold reward member status buying electronics in the winter, returning customers with silver reward member status buying electronics in the fall, returning customers with bronze member status buying electronics in the spring. All cases with `User Status` equal to `New` received discounts when they should not have. Cases with `Rating` below the valid range did not generate the proper error messages. `Rating` was not robust to integer overflow or underflow. Cases with `Age` below the valid range did not generate the proper error message. `Age` could be set arbitrarily high (which could be considered a validation failure) but was not robust to integer overflow. Names containing an underscore did not trigger an appropriate error message. Names of length greater than 10 failed to generate the proper error message. All told, 12 well established defects were discovered during testing.

D. *Conclusion*

This was an individual project, with myself as the sole contributor. Through the execution of this project, I gained proficiency in designing test cases using equivalence partitioning, boundary value analysis, and cause and effect testing. These skills will be instrumental in my aspirations for future solo development and testing.

II. Project 1 Part 2: Design Of Experiments

A. *Introduction*

Broadly, design of experiments (DOE) refers to the practices by which controlled tests are designed, coordinated, and interpreted[1]. This project narrows its focus by providing an implicit hypothesis; a mobile application and suite of tests has been designed, and it must be determined whether the application will function on a variety of models and in a variety of circumstances.

To have complete confidence in the application's functionality, each test would need to be repeated in every unique combination of circumstances, this is called full factorial design. To design a set of circumstances that allows for a reasonable level of confidence without succumbing to test bloat, pairwise combination testing is employed. This method is based on the idea that most defects in software arise from the interaction between no more than two factors[2], and operationalizes this theory by testing all possible pairs of parameters.

Presented with 5 parameters to consider, `Type of Phone`, `Parallel Tasks Running`, `Connectivity`, `Memory`, and `Battery Level`, a tool employing pairwise combination testing is selected and assessed on its ability to generate a minimized set of tests circumstances that provide pairwise combinatorial coverage.

B. *Solution*

The tool selected was Pairwise Pict Online[3], a minimal online application powered by Microsoft Pict that is explicitly purposed to perform pairwise combination testing with restraints. The user interface mainly consists of an input text box initially containing an instructional example, and an output text box, where the pairwise combinations are displayed. A `Generate` button present between the fields runs the program. Two hyperlinks at the bottom of the page enable downloading the test factors and generated test cases as .txt files. This tool was selected due to its minimalistic and intuitive nature, and because its simplicity and narrow focus match the scope of the problem.

C. *Results*

The tool performed well, successfully minimizing the number of test cases. The 27 tests generated offered complete pairwise combinatorial coverage. For comparison, full factorial testing would have resulted in 800 tests. Because the largest 2 parameters each contained only 5 discrete options, it would be impossible to have less than 25 pairwise combination tests. Considering the multitude of other factors, it is likely that the 26th and 27th test cases were necessary for complete coverage. Since no restraints were given, this part of the tool's functionality was not assessed.

D. *Conclusion*

This was an individual project, with myself as the sole contributor. Through the execution of this project, I added Pairwise Pict Online to my arsenal of testing tools and gained firsthand understanding of the importance of employing DOE as a strategy to minimize testing bloat.

III. Project 2: Structural-Based Testing

A. *Introduction*

Structural-based, or white-box testing, is a methodology based on the derivation of tests from knowledge of an application's internal implementation and architecture. This project explores two domains of structural-based testing, code coverage analysis and static analysis.

Code coverage analysis specifies a hierarchy of coverage. Each tier includes the preceding tiers and represents a greater degree of certainty that every error will be discovered. In ascending order, the tiers are statement, branch, path, and condition coverage.

Full, or 100% statement coverage indicates that each statement in the code is executed by the test suite at least a single time. Extending that, full branch coverage indicates every route through every decision point in the code is tested one or more times. Path coverage builds upon branch coverage, ensuring that every combination of routes through the code is tested once. Condition coverage further specifies that every combination of predicate valuations within each decision point should be tested at least once.

Static analysis, on the other hand, refers to the analysis of code without the context provided by test cases. This type of analysis is largely variable and tool dependent, with each tool having a slightly or largely different focal set of coding patterns to identify and flag.

The project involves two parts; the first is to design a suite of tests for a vending machine application, select a code coverage tool, and attempt to achieve 100% statement and 90% branch coverage. The second is to select a static analysis tool to use on a shipping cost calculation application. Both selected tools are evaluated for efficacy.

B. *Solution*

IntelliJ IDEA Code Coverage Runner was paired with JUnit tests to evaluate the vending machine application. The tool comes pre-installed with IntelliJ IDE and provides statement and branch coverage.

Four categories of tests were developed for this application: Tests wherein the user provides more than enough money, exactly enough money, less than enough money, or specifies a nonexistent item.

For the shipping cost calculation application, PMD plugin for IntelliJ IDE was used to perform static analysis. The plugin evaluated the code on predefined categories including best practices, code style, design, documentation, error prone, multithreading, and performance.

C. *Results*

According to IntelliJ Code Coverage Runner, a total of 11 tests split amongst the 4 categories provided 100% statement coverage and 93% branch coverage of the vending machine application. The last category, wherein a nonexistent item was specified, contained only one test case; this was the singular failing case, and indicates a potential validation failure or that the application is intended to be given names from a set. 100% branch coverage was unable to be achieved due to a nested `if` statement making one of the branches impossible to reach. Code Coverage runner was straightforward and effective, providing exactly the necessary coverage and requiring very little time to set up and learn to use competently.

Notable anomalies detected by the PMD plugin in the shipping cost calculation application included improper string comparison using == and a lack of braces on one of the `if/else` statements. Good recommendations provided by the tool included avoiding literals in `if` conditions, including the class in a package, including comments for each method, discouraging the unused local assignments of cost and output, discouraging the use of `system.out.println` in the finalized codebase, encouraging finality for some local variables and method arguments, and urging that the codebase be constructed as a utility class.

The tool did not seem to take issue with the non-functional constructor, except to state that the variables therein were unused. It also did not find fault with the redundancy of calling `toString()` when concatenating an integer and string in java, or with the unsaved/unprinted first call to `calculateCost()`.

Overall, it is difficult to be excessively critical of the PMD plugin, as it allows users to define custom rulesets. Additionally predefined flags are linked to more thorough descriptions on the PMD website in case the rather short error descriptions are insufficient for assessing whether a fix is necessary. The tool is very easy to set up and appears to have near infinite potential for users interested in customizing their static code analysis.

D. *Conclusion*

This was an individual project, with myself as the sole contributor. Through the execution of this project, I added the IntelliJ Code Coverage Runner and the PMD plugin to my arsenal of testing tools and discovered the potential benefits of defining custom rulesets for static code analysis.

IV. Project 3: Graphical User Interface Testing

A. *Introduction*

Graphical User Interface (GUI) Testing is another broad slice of the verification and validation paradigm and can be divided into functional and non-functional components. Non-functional GUI testing encompasses many sub-categories of GUI testing such as compatibility testing, accessibility testing, security testing, and usability testing. Functional testing is concerned with verifying the correctness of all the interactive elements of a GUI as they function in the application, and serves as the focal point of this project.

The goal of this project is to select a tool and use it to perform automated functional GUI testing on 2 distinct versions of some application with a minimum of 3 pages. The original version requires the inclusion of various specified GUI elements, and the second version requires that the flow of pages be altered, and at least 3 elements of the GUI be changed in orientation, size, or location. Once implemented, the selected tool is utilized to design a test suite for the original version which is then run on both the original and modified versions. The tool is then assessed, with special consideration for the comparison of the test case outcomes on each version of the application.

B. *Solution*

A react quiz application was created to satisfy the given requirements, and a second version was created with an appropriately modified GUI. The Selenium[5] web extension was selected as the automated GUI testing tool and used to design 8 test cases on the original version. The tests were run on both versions.

C. *Results*

On the original version, 8/8 tests completed successfully, resulting in 100% statement and decision coverage. The same tests were run on the modified version, which resulted in 4/8 successes. When unable to find an element necessary to the testing process via CSS nth child selection, Selenium attempted to use XPath to mixed results. The tests necessitating the use of secondary selectors took much longer, as Selenium does not appear to be optimized for this type of location. Selenium was able to successfully select a relocated button in 4 cases but failed due to timeout in 2 cases. The other 2 failures were caused by the changes in page flow.

From this use case it is apparent that selenium is a powerful tool for web-based GUI testing. The set-up was fast and straightforward, and the test case recording feature was easy to learn and use. Of particular note was Selenium's robustness to altered elements, though not perfect in that regard, it still produced noteworthy test case reusability of 50% between versions.

D. *Conclusion*

This was an individual project, with myself as the sole contributor. Through the development and execution of this project I was able to incorporate React into my development toolkit, and gained familiarity with Selenium as it applies to automated GUI testing.

References

[1] S. C. Reid, “An empirical analysis of equivalence partitioning, boundary value analysis and random testing,” *Proceedings Fourth International Software Metrics Symposium*. doi:10.1109/metric.1997.637166

[2] “What is design of experiments (DOE)?,” 4.3.1. what is design of experiments (DOE)?, https://www.itl.nist.gov/div898/handbook/pmd/section3/pmd31.htm (accessed Nov. 6, 2023).

[3] C. B. Monteiro, L. A. Dias, and A. M. Cunha, “A case study on Pairwise Testing Application,” *2014 11th International Conference on Information Technology: New Generations*, 2014. doi:10.1109/itng.2014.42

[4] Pairwise Pict Online, https://pairwise.yuuniworks.com/ (accessed Nov. 6, 2023).

[5] “The selenium browser automation project,” Selenium, https://www.selenium.dev/documentation/ (accessed Nov. 6, 2023).