**SE463 Assignment #6 Report**

Building & Organizing Test Cases: Comparing The Experience of Similar Programs From a Black-Box Testing Perspective

By: Elizabeth Dixon

**Table of Contents**

* Pg. 3: Project Overview
* Pg. 4-6: Test Case Organization & Outcome Analysis
* Pg. 7: Project Reflection

**Project Overview**

Building test cases is a skill that at times seems self-explanatory but can also seem vast and daunting on the other side of the spectrum. However, testing a program’s experience from a black-box perspective can eliminate some of the uncertainty as the tester is put in the end user’s shoes. Therefore, the tester can recognize certain aspects of the program that could be altered and improved that may not otherwise be recognized from the perspective of a developer or white-box tester. Although a black-box tester's test cases may also leave out some information that a white-box tester would be able to employ, a program such as the one provided to us for this project is a great example of all that is possible for a system to be tested on from an outside perspective. Although both build\_1 and build\_2 have their pros and cons and can both be improved, it was beneficial to see how a similar or exact same set of test cases yielded different outcomes. Additionally, it was interesting to see how the chosen process by the tester (in terms of testing each program) can influence how the other is tested. To begin, I thought that it would be easiest to create test cases based on the program requirements. This is not only because they are essential, but also because writing them out and testing them first would inform me of the semantics of the program better. Then, I would be able to create supplemental test cases that would provide a more comprehensive overview of the behavior of these programs and, therefore, what could be improved from a development perspective. Additionally, throughout the whole test case-writing process, I found myself thinking in a BVA or Unit Testing manner, as I had to consider how an input would act if it was entered as intended by the program, but also how it would act if a possibly wrong input was entered.

**Test Cases Organization & Outcome Analysis**

As previously mentioned, I began with writing out test cases based on the program requirements first but ended up writing supplemental test cases along the way that also related to the program requirements. First, I started by writing test cases that related to running the program then what I would know as a user based on what the program told me. For example, "Does build\_x run on my OS environment on Terminal?”, “Based on the directions… I should do this”, “Based on the directions… I should do this. But what if I did this instead?”. However, even at this point, I fear that I would still have some questions about how to use the program as a user. Both programs give a line of dialogue that indicates that the program is running, but there are pros and cons to each line. Assuming that I would know that this is a calculator program (especially based on build\_1's dialogue), I would have somewhat of an idea of how to enter values to operate on. However, I wouldn't know what my options for operators would be because the program does not tell the user so - I would only be able to make assumptions. Plus, if I didn't know ˆC to quit, I wouldn't know how to quit the program for build\_1. However, for build\_2, it *does* specify how to quit but still does not tell the user what their options for operators are. Another flaw for build\_2 is that it does not tell the user a critical piece of input formatting - that the tokens have to have spaces in between them (but build\_1 does). That can somewhat be assumed through the “Please input your formatted infix expression.”, but it is still not entirely clear. Additionally, what if the user does not know what a "token” or a “formatted infix expression” is? The terminology being displayed to the user is also something to consider - in some ways, more is more for the UI of a simple program such as this.

Next, I focused on the semantics of the operators - mostly with the considerations of “Can the program handle operator \_?” and “Does the program recognize the operator \_ being used in an invalid way as invalid?” with various different combinations of inputs and sequences of inputs for one run of the build. I would say that build\_1 was harder to test in this section. Particularly because it became clear early on that the conditionals for what some valid expressions were were not fully fleshed out. When the formatting specified by the program was followed exactly by the user, it more often than not acted as it was supposed to. However, it did not output “Invalid input. Please try again.” nearly as often as it should've when testing invalid sequences of expressions. For example, it would give an output for something that the program is not meant to handle and/or it wouldn't specify what the issue was when there was something invalid, possibly making a simple mistake unclear for the user. As for build\_2, it did a much better job at checking for correct or incorrect syntax and would even tell the user what part of the expression yielded the “Incorrect input” output and what the category of incorrectness was. Additionally, it would tell the user what the solution was which was nice. One very telling test case that I provided for both builds for testing the ‘%’ operator was something that would yield a “Divide by 0” error so that I could see whether or not each build had proper error-handling or not. build\_1 did not and “Abort (core dumped)” the run whereas build\_2 did not have a core dump and instead provided the user with a specific “Invalid input - Division by 0 prohibited.” message and continued to loop through the program as intended.

Additionally, I thought of a few supplemental tests to run such as “Since the program can use the '( , )' well, can it recognize PEMDAS rules at a high level?” and “Does the program let the user quit using ^C?”. Both builds had pretty different behavior for each of the test cases with build\_2 acting more appropriately once again. For the PEMDAS test case, I used one of the infamous expressions of “8 / 2(2+2)” and wrote the inputs in many different ways to see how each build would react. build\_1 would give outputs to certain expressions that should not have yielded outputs (mostly with 0s) while build\_2 would give “Invalid input” explanations. Interestingly, both programs acted correctly when I specified the ‘\*’ operator in between the 2 and (2+2) parts of the expression so I think that the assumption of 2(2+2) yielded multiplication is just not in the nature of the program. As for the quitting using ^C tests, build\_1 allowed it whereas build\_2 did not. Although it's convenient for it to be allowed in build\_1 it is a good indicator of thoroughly checked security in build\_2's case. However, build\_2 would have to run perfectly (which it appears it does…) without any semantics issues for this to be appropriate - otherwise, the user would be stuck and would have to quit Terminal (or whichever medium they're using) to quit the program.

**Please refer to the eadixon/se361\_fall2024/A6 repository on GitHub to view a complete spreadsheet of test cases and pictures of executable run evidence.**

**Project Reflection**

Overall, as mentioned in the introduction, I felt that going through the process of building these test cases then testing them thoroughly was a great learning experience that'll help me carry out testing processes better each time I do it. Repetition will always help, but understanding the semantics of each test case is even more helpful and will inform me in testing future programs that act in a similar manner as build\_1 and build\_2. I liked that I was able to use this experience as a way of explaining how and why a program works in detail - as well as how it should and shouldn't operate. I could imagine giving a report on a program this way out in industry if needed. It reminds me of how I was tasked with researching timestamping tools for my Butler IT internship and how I wrote a detailed report on said tools. Although it's not exactly the same thing as unit and system testing a simple calculator program, they both yield lots of detail and explanations that could save people time when they would otherwise have to do their own extensive research. I also think that it's useful to realize, from an outside perspective, where something would need to be changed in the source code - it would interesting to actually be on a team where I would be able to inform a developer of a certain lapse in program behavior that they would then come up with a solution for based on my tester-perspective findings.