Pomai Ahlo

SNHU 21EW2

CS 320 – Professor Wilson

December 10, 2021

Summary and Reflections Report

In this project, I acted as a software engineer at Grand Strand Systems and was tasked with creating a mobile application for a customer. For this scenario, I created and tested contact, tasks, and appointment services. The purpose of these services was to add, update, and delete contacts, tasks, and services within the application.

My testing approach was robust and aligned with the software requirements well. I tested each requirement twice – once to confirm the method worked as expected when all arguments were legal, and once to confirm an error was thrown when there were illegal arguments. For example, the contact object should have a required firstName field that cannot be longer than 10 characters or be null. I tested when the firstName met that requirement, and when it didn’t. These were small programs that it was possible and within reason to test every type of erroneous input. In the above example, an illegal first name could either be null or longer than 10 characters. I made sure to test both null and lengthy inputs.

High coverage rates are favorable. As our reading states, “a test coverage rate of 80% or above is considered reasonable.” I tested every function, yielding a 100% test coverage rate. I did this because it was necessary, as functions that we are usually able to skip were written such that they still had to be tested. Namely, setter functions had checks in them. Getter functions were also checked indirectly as they were needed to run other tests.

In covering every possibility, I ensured the code was technically correct. Compared to the first time I submitted the assignment, I found a minor defect, which is a good sign. I also made sure the code was technically correct by assessing what could go wrong and testing to make sure those things didn’t go wrong. For example, in my TaskService class, the program iterates through a list of tasks to search for a specified task. So what I did was make sure I tested a list with multiple items to iterate through, and I tested modifying an item in the middle, making sure there were not any logical errors.

The only thing I did not check was time efficiency. I know now that I could have appended to my tests “(timeout = x)”, where x is the number of milliseconds the test could take before being flagged as a failure. This is better than manually checking the length of time it takes to run a test, as that is a tedious task. I chose not to do this because there were no requirements for performance, but in the real world, there probably will be such requirements.

The three milestones had pretty similar requirements. Because of this, the structure of both the code and the tests was structured in similar ways, and they all used the same software testing techniques.

Junit tests are considered dynamic testing, as the code is executed. The primary technique used was a specification technique called equivalence partitioning. Equivalence partitioning involves splitting data into valid or non-valid categories.

A proper test of a valid input suggests that other valid input should yield the same results, and the same applies to invalid input. There was also a little boundary value analysis. Boundary value analysis tests the boundaries of the invalid and valid range. In the contact service tests, there was the opportunity to test different kinds of invalid and invalid input, and some input near the boundaries were tested. Structure-based testing in the form of statement coverage was done when test coverage was determined.

No experience-based testing was done, as I have little experience as a tester. Experienced-based testing techniques rely on the experience of the tester. For example, there could be error guessing, where the tester makes an educated guess on where the system might have a weakness. This can make testing more efficient since the tester’s experience makes it easier to identify what doesn’t need to be tested and what should be tested extensively.

There also weren’t any decision tables, which are commonly used to test combinations of inputs. Inputs in this case did not cause much variation, as one invalid input would always cause an error to be thrown. There was also no need to use case-based testing or state transition testing, because of how the code was structured.

When I revisited my code, I did so cautiously. This led to me adding tests to check every type of wrong input. As mentioned before, I found a minor defect in the process. This was found in my Contact class, where I did not check if the phone number was null. It did still throw an error, but not the right kind of error, which would have potentially made things confusing for anyone debugging if they ever ran into that issue. This shows that it is important to understand and appreciate the complexity of the code.

As the developer, I was very arrogant the first time I turned in these assignments. When I first did the TaskService class, I only tested changes to a task list that had one task to it. I didn’t realize that my code would not work if there were several tasks already being implemented. I also did not notice that there was a test that passed that didn’t test the method in question. When I returned to these assignments to tidy up for the project, I came in with a better mindset, one that didn’t think that I had already done everything right. Only then could I find my mistakes, and not make them hard.

This project required discipline. There were a lot of aspects to it that were seemingly repetitive, but not truly so. Cutting corners only created tech debt for me, as I had to rewrite some tests, and in some cases, write completely new tests that I didn’t have before. Had I had discipline from the start, I would not have had to deal with all of that.

References

García, Boni. (2017). Mastering software testing with JUnit 5. Packt Publishing.

*Software Testing Techniques.* (Mar 2021). Geeks for Geeks.

https://www.geeksforgeeks.org/software-testing-techniques/