Brianna McCollum

Summary and Reflections

The communication app implemented three different objects and a servicing class for each: Contacts, Tasks, and Appointments. All three of the functionalities for the communication app had a set of JUnit tests focused on similar but not exactly the same testing criteria. The varying test criteria were determined by the software requirements and goals of each object in terms of their functionality for the communication app. For instance, Contacts had differing information stored in them than Tasks and Appointments, needing a first name, last name, phone number, and address. Understandable things that a realistic contact would have to identify the person and their contact information. These data points are of different lengths and types than the data held in the other two objects, and therefore the Contact class needs different tests to verify its functionality.

The tests had a pretty high coverage percentage overall, around 85% for each of the respective classes that they covered. While not flawless coverage, complete and total coverage is an unrealistic standard. High-quality testing aims to cover as much as possible with as little time and resources as possible, full coverage of a real-world project would be near impossible due to all the fringe cases and large amounts of interconnection between different programs. The coverage percentage alone doesn’t outright tell you that the tests were effective, but comparing it to the requirements can paint a greater picture of the code’s functionality. Each test block tested a class, not a specific function, but rather a specific requirement of the software. If that test block passed, that meant that the requirement was covered, which means the high percentage of coverage translated to a lot of not all of the requirements being satisfied.

The JUnit tests were both technically sound and efficient in their creation for a few different reasons. They used the JUnit test format to their best, utilizing statements like assertTrue, assertFalse, and mostly assertEquals to assure proper functionality from the different classes. These tests aim to replicate user stories, or rather things that realistically the user would try to do with the program. The tests interpret these user stories into JUnit tests by breaking down the requirements into programmatic pieces and then referencing each of these pieces. Some special functions had to be created outside of the requirements, such as the isEmpty function in each of the service classes. This function is simple, checking if the service’s list of objects is empty and returning true or false. It’s not used for any requirements or any of the user’s functions, but rather just to verify that deletion takes place successfully. The tests only do this when necessary, and only use the bare minimum code to achieve their goals. For instance, the Appointment Service JUnit test class only has a single test block. It tests adding and removing appointments, which are the only requirements of the appointment services, making this simple block incredibly efficient for testing.

As stated previously, I used requirement-based user stories as the foundation for developing my tests. With the general goal of my tests then designed, I altered them a bit to make them more streamlined and efficient. All of my tests were Dynamic testing, which is testing that involves executing the code to see outputs and certain behaviors. This allowed for a wide variety of inputs to be tested so I could check the fringe cases for each communication app function. I split up these checks using something similar to equivalence partitioning, where each block of JUnit tests focused on an “equivalent” group of inputs. In the case of the Task class, one test would be the average input, another would use empty inputs, and another would try inputs that are too long. This is a great method to use when testing a program with a lot of fringe cases or a wide variety of possible inputs, as a single test covers many different possible inputs.

I utilized a small amount of Static testing while testing on the communication app, and that was just self-inspection of my own code before constructing the JUnit tests. Static testing involves testing without executing any code, things like peer reviews and formal inspections. I combed through my own code at the beginning of the project to make sure there weren’t any big issues, but then the rest of my code was Dynamically tested. Static testing is excellent for large-scale projects or projects where a lot of concepts are written out before being implemented into code. It can serve as a very helpful tool for preventing failures, especially when combined with paperwork by spotting defects before they’re even written.

The mindset of a tester is very different from the mindset of a software developer, using a different set of skills to locate and address possible errors in the code. For this mindset, caution is a very important trait to employ. It can be so easy to simply plug in tests until the quota of coverage is met, but then so many possible problems and fringe errors will be unfound and shipped in the final product. When tackling a set of tests, there must be a level of scrutiny. Errors aren’t lurking just in the code, but can also exist in unwritten interactions between pieces of code or even outside compatibilities. An error may exist in the way that a service class refers to one of the communication objects, but the individual code in both classes is objectively correct. Problems like these require caution, pausing for thought, and being careful and thorough.

Originally, I didn’t think bias would be a problem for my code. Bias is usually used to discuss more emotional concepts like writing and debate, not for computer programming. However, I discovered through testing my code that things I found previously very sound ended up being a bit shaky later on. It took me quite some time, in the beginning, to make tests because I kept assuming that certain parts of my code were functional when they weren’t. But once I realized what I was doing, I limited my bias by always assuming there was a better way to write the code. It helped a lot, and my tests immediately improved.

Discipline is important no matter what field you’re aiming to work in. Discipline keeps me on track, it regulates the quality of my code since I keep myself accountable and held to a certain standard. Without discipline, my work would be sloppy and rushed, keeping me far from the bar of quality I need to achieve to be a successful software engineer. Cutting corners while writing or testing code is basically signing yourself up for problems in the future, as those cut corners will no doubt cause problems that at best is inefficient and at work will cause failures. By cutting corners in software development, I’d be forking a ton of technical debt onto myself or my poor testers. By quickly throwing together a piece of code for the communication app, it will cause issues in the testing and need to be fixed later, which will ultimately just take more time and resources to deal with than doing it right in the first place. It’s important to be disciplined and put in your best effort on the first try. It’s ok if you mess up, accidents happen, but by focusing and keeping myself to a higher standard I can minimize slip-ups and make my code better.