JUnit Tests Report

Throughout this course, I was challenged to verify the quality and correctness of my code by writing tests. As testing is a crucial part of the software development process, it’s important for developers to know how to test their own code. Personally, I found the process of using JUnit to test my code to be interesting and worthwhile.

My tests were written specifically to satisfy the software requirements. Whenever writing a test, I would reference the rubric, making sure that each test tested a concrete detail within the rubric. For example, whenever it was necessary that a field not be null, I wrote a test to specifically check that passing a null value would result in an exception. Similarly, I checked to ensure that Strings did not surpass the set length given by the software requirements.

I am confident in the quality of my tests because I was careful to write simple, specific tests. Additionally, I made sure that my tests covered a large portion of the code. Easily over 80% of executable code was covered during testing. This is a reasonable amount of code coverage for most projects, and certainly for the Project One assignment.

I ensured my code was efficient by minimizing the number of lines of code and by only allocating memory when required. For example, in the TaskTest class, I only created a new Task object when needed. Although this assignment did not require me to use memory or processing time conservatively, this is no reason to write badly-performing code.

I employed a variety of software testing techniques when testing my classes. I used black-box testing most prominently, as it was the easiest way to ensure I met the software requirements. It was a priority that all black-box tests were written before looking into the structure of my code. By purely looking at the test conditions and the output from my program, I could avoid the complications of systematically testing my program’s structure until absolutely necessary. I find it intuitive to write code that works first and foremost, and then to revise the structure and add optimizations later.

White-box testing came in handy when black-box tests failed. When black-box tests failed, I had to look at the internal structure of my code in order to ensure my program worked. Making sure I had a wide test coverage greatly helped in testing my programs. Whenever an important piece of code was not being tested, I made sure to write a test function to cover that code. This is where JUnit’s features really came in handy; the automatic color highlighting done after testing made it obvious what features were being tested successfully and which ones were being missed. For example, JUnit immediately let me know that the addAppointment() function in AppointmentServiceTest was initially not running. I would imagine these features would be even more useful for a long, complicated software project.

One testing technique that I did not perform was having a formal review of my code. Although I was casually reviewing my code as I was writing it, and I would often review the code again before executing it, I did not have anyone else look at my code. For a larger, more complex project, it would be a necessity to have another programmer or computer scientist review my code for errors and improvements.

In my opinion, a healthy mindset is crucial to being a good programmer and, in general, to being successful in life. When it came to testing, my mindset revolved around testing extensively while editing my code only when necessary. A deal of caution has to be used when testing code, as one does not want to break everything that has been built by invasive testing. However, sometimes, the image of a structurally-sound piece of software is an illusion, and testing helps to illuminate where this software actually fails. I was careful to test only want I specifically wanted to be tested (such as adding and removing functionality in the service classes), and I was diligent in performing tests which would ensure I meet the software requirements.

As the developer of my code, I have a clearly biased view of it. One obvious example of this is that I have an information bias: I know more about my code and how it works than anyone else, because I was the one who wrote it. Although this makes it easy for me to write tests I want to write, it makes it difficult to think outside of the box and create tests for diverse, unexpected situations. It also means that there may be glaring mistakes in my code that I am oblivious too. Although the classes we wrote in this course were simple, this bias becomes a huge problem when writing complex software. For a larger project, I would want other people to review my software and do dynamic tests on it in addition to my own testing.

Discipline is an incredibly important facet of any skill. When it comes to testing, to be a disciplined tester and programmer is to write simple tests with a wide code coverage. It means to test every important piece of your program and to consider a variety of inputs and possible ways in which your program might break. In many situations, the domain of possible inputs is virtually infinite, such as the number of possible 50-character strings, so it is not feasible to individually test each input. However, a disciplined tester will do the necessary equivalence portioning and write the necessary tests to ensure that the program can handle a 50-character string from the user. For a solid example of how I can be a more disciplined tester, I will try to always test when receiving an input string from the user. I will test the case when the string is null and when the string should be successfully accepted. I will also test the case when the string is above the character limit, when a character limit is present.