Travis Garlick

6/December/2021

CS320

Project 2

**Summary**

In the first two modules of Project 1, I completed two very similar milestones. Beyond the code within the service module classes themselves, the purpose of these two assignments was to demonstrate the concept of testing code. Because of that, I will first focus on the JUnit testing portions of the two milestones.

The parameters of the elements within each class in the code were simple and well defined, so I would say that my approach to testing was to completely cover the software requirements. Each variable was defined to be within a certain length and not be null, so the first step of testing was to demonstrate that any input outside of those parameters would not work. The “phone” variable, for example required three tests (not including the initial valid input test): A test for a phone number that is too long, a test for a phone number that is too short, and a test for a missing (‘null’) phone number.

The Eclipse IDE has convenient tools for tracking the coverage of JUnit tests against the total code. Running the tests not only show which tests passed or failed, but also the order the tests were performed, and even includes a percentage showing how much of the code has been tested. Even without being able to use these tools on completed code, the software in these two modules has few enough conditions that I can ensure they are being tested by visual inspection alone. There are 8 string variables, each with a specific required size range of characters. After that, there are 6 methods so far for adding, deleting, or changing items in two different lists. This brings the total number of objects that need to be tested up to just over a dozen.

By following the instructional video that we were provided in module 3 for JUnit testing, I can ensure that my code is technically sound. For example, I now know it is helpful to throw exceptions in the executable code as an error handling method that can aid in the testing process, such as in the lines:

if(ID == null || ID.length() > 10) {

throw new IllegalArgumentException("Invalid ID");

}

JUnit tests also provide a function that shows how long a test takes to perform which can be used as a benchmark to monitor one area of code efficiency. To write efficient code, there are many methods a programmer to use. One such method I used in these two assignments was to only test the boundary conditions. For example, the “Contact ID” variable must be a non-null string of no more than 10 characters, so (aside from the null test) that variable only needs one test beyond the boundary condition:

void testContactIDTooLong() {

where I set the contact ID as the 11-character string "12345678910". If an exception for an input that is too large is thrown here, it is safe to assume that any other ‘out of bounds’ input (like “1234567891011”) would also follow the same rule, so further testing of that variable would be unnecessary and inefficient.

**Reflection**

The three milestones for Project 1 all involved software testing techniques using JUnit testing.

Milestones one and two both used identical testing techniques. The first of these techniques involves error handling within a class of the program for each failure condition. For the purposes of these assignments, that specifically means three lines of code for each variable that will throw an error prompt if that variable is null, or not the correct size (assuming the variable has size limitations). Each of these thrown errors are to be checked by JUnit tests.

This leads to the next step in the process, which is creating JUnit test classes for each class of the software. This generally starts with a test case that checks whether a set of correctly formatted test variables can be assigned properly. From there each variable is checked to determine that incorrectly formatted variables throw an expected error. A separate set of JUnit tests are used to test the proper functionality of any callable method in the software, such as adding items to a list, or deleting items from a list.

The third milestone of this project introduced a new testing technique involving checking dates. The software for creating appointments requires the ability to check that the date for an upcoming appointment is not in the past. Just like before, if such a case does occur, an error should be thrown and checked by a JUnit test case. While the software produced in these milestones is not a real product being made for a client, the techniques used do represent useful real-world applications for a system that would need to manage a list of contacts or set appointment dates.

In these milestone assignments, we did not have to use any test documentation techniques. Not counting these journal entries, or the brief comments within the code itself, I have made no record or explanation of the testing process like a document that might be submitted to a project manager for a company’s software product. If I were to write such a document it would likely provide an overview of all the tests, with an explanation of the results, how much of the code is being tested, and perhaps speculation about errors that might still occur in the software that wouldn’t be found by these tests. This project has also not allowed us to employ the practice of having the tester and software developer be two different people. Such a situation might be more desirable for organizations with sufficient staff that want a tester to approach the software with a fresher, more objective perspective than the software developers themselves might have.

My mindset in this project is admittedly different than one I would have in a job in the “real world”. This software is not a real product indented for use by customers or clients. If it were, there would be greater consequences for defective code and, I would have had to approach the software design with far more caution and discipline. For example, I might have needed to eliminate every basic error from each method before submitting my milestones, to ensure that they will at least compile for the purposes of dynamic testing.

Even after getting as far as writing runnable code, I would still have a considerable amount of personal bias in it as both the developer and the tester. To avoid falling into the trap of overlooking defects in a program that I am familiar with by design, it might have been a good idea to find someone else unrelated to the development process of this code. Then I could have them review my work with a fresh pair of eyes.