Russell Pallas

February 20, 2022

CS320

Project 2

I cannot verily easily summarize my approach to writing and performing unit tests into one specific category or mindset. I tend to take a multi-faceted approach when it comes to various tasks that I must perform and determine which of those approached best accomplishes the specific task I must perform. When it comes to writing and performing the unit tests for the codebases that we wrote I took a more dynamic approach. I followed the software requirements that we were given in a bulleted list. One-by-one I wrote the code required to meet those requirements. My approach to writing the code required for these requirements was dynamic in the sense that I allowed myself to compile and run my codebase as I wrote it. I find that this approach allows me to fine-tune anything in the code that might allow me to better meet the requirements or make my code more efficient and more readable. This allows me to simplify the code that I will have to read through and understand completely during my static testing. This is where I will read through all of the code that I have written and determine whether or not it not only makes sense, but also I will determine whether or not I could optimize the code that I have written. Within the task manager that we had written, I found that by following this procedure, I was better able to write the “addTask“method.

One thing that I did not use for these projects was something called “Infinitest”. This would have allowed me to perform the tests that I had written as I write the code. I believe that this would have been beneficial when it came to writing not only some of the tests that I needed to perform, but, it would have also helped me to write and optimize the methods that were required for the various classes. I also found myself performing less static testing than I likely should have performed. However, I find that although a more dynamic approach is not an alternative to static testing, it tends to accomplish a similar goal when it comes to getting methods written for certain classes. I will most certainly use Infinitest in the future when I have more time to familiarize myself with it and learn how to best use it in writing Junit tests. Although by not using Infinitest I was required to perform more work to ensure the most code coverage from my Junit tests as possible, I was still able to accomplish the most coverage that I could with the methods that I had.

When it comes to writing the Junit tests, I found that they were sound by running the tests in a dynamic fashion with code coverage. For example, when it came to testing the class tests (i.e. task accessors/ modifiers, attributes, and other methods), I would modify the tests to ensure that the entire class was green, including all “if” branches. One helpful piece of advice that I was given was keeping logic out of the constructor. I put all of the requirements for the attributes (i.e. id, name, address, etc. length) in the modifiers instead of in the constructor itself. By doing so, I was able to test the constructor and the modifiers would also be tested instead of having to test the constructor and modifiers separately. This allowed me to be far more efficient in my method writing as well as my Junit test writing.

Again, the most used software testing technique that I used was dynamic testing. I find that the test-as-you-go approach works best for me when I know what I am doing. I also used “peer” (instructor) review to write better methods as well as Junit tests. I believe that for the student, this is one of the better software testing techniques that you can follow. I find myself to learn by example, so when I can write a certain method or Junit test and it is picked apart by someone else, I will learn from that example that I received feedback on. I do not necessarily believe that this would make for a strong career in testing, however, as a student I am here to learn from my mistakes. I certainly learned from my mistakes and the feedback I received on them. Dynamic testing is useful in all situations where you are able to compile as you wish to ensure that certain aspects of your code are functioning as expected. This might be especially usedful when it comes to some sort of graphical user interface that you would like to look a certain way. Taking openGL as an example, you would be able to continuously run your code to ensure that a certain object is where you would like it to be as well as the shading and color of that object. Static testing will also help with this. Using the openGL example again, if you had a triangle with three points and you would like the first point to be red, the second to be green, and the third to be blue, you would want your color array to look like (255, 0, 0, 0, 255, 0, 0, 0, 255). However if you had switched one of those numbers a static test (reading thorugh the code prior to testing) would likely show that those numbers are swapped.

My mindset when it came to these projects was something that I am not used to, but was able to adjust to. This term I found myself not putting in as much effort as I typically do. With my new position at work, I find myself with less time to complete some of the learning that I would usually do prior to even starting to thing about any codebase or Junit tests that I will be writing. This might seem somewhat obvious, but I find that when I know what I am doing when it comes to writing code that things tend to go much smoother. When it comes to these projects, I found myself learning as I went along. This is not a good mindset or set of behaviors to have when learning a new concept. After my first milestone, I found myself being more cautious of whther or not I was testing the methods of the classes themselves or testing whether the methods were successful. For example, in the first milestone I was actually writing tests for essentially the ArrayList class rather than my Contact and ContactService classes. However, one thing that I did take into consideration was how all of the classes and their methods would work together. This means that I was able to call methods from the Contact class in the ContactService class. This allowed me to limit the amount of code that I had to essentially rewrite. For example, when it came to updating contacts, rather than creating a whole method to update certain parts of a contact, I was able to just call my modifiers from the Contact class within the ContactService class.

I feel as though testing your own code is beneficial in the sense that you know what you wrote within the classes and therefore you have the best understanding of the relationships between classes as well as the classes themselves. However, good code is code that someone with no exposure to your program could look at and understand. This means that you should be able to write code that someone else could easily write the Junit tests for. So, although it is beneficial to test your own code, you should be aware that you might change some aspects of the classes to meet your tests. This could mean that you now fail to meet a certain requirement. On the other side of this, you might change a test to accomplish full code coverage, which might also mean that you could fail to meet the requirements given to you.

Cutting corners could be a major concern when it comes to writing code. Although it might make the present moment seem to be a little easier, it will more than likely come back to haunt you some time later down the line. Avoiding cutting corners correctly requires discipline in the sense that you must cut them just enough to not avoid any catastrophic bugs in your software. Copying and pasting code is a prime example of this. There is nothing saying that you cannot copy and paste code from another project of yours into a project that is similar, however, you must perform both a static and dynamic test of this code to ensure that it works with the project as a whole.