# CS 320 7-2 Project Two

Taylor Anderson

[taylor.anderson5@shnu.edu](mailto:taylor.anderson5@shnu.edu)



# Summary

Approach

As I mentioned previously in my 4-2 Journal entry, I have come to learn what a testing approach is rather than simply repeating a definition. I think this shows an internalization of the information and a far better understanding rather than simply citing a definition. With that said, in my previous journal, I mentioned, “I would define the testing approach as an approach to programming that sets clear tests of the software from the beginning to ensure the program meets the requirements as it is being built.” For this assignment, I had to put all of my previous assignments together, with the goal of reaching at least 80% coverage. While each of my previous assignments ran successfully with their intended purpose, when I ran the coverage, I was originally well below the 80% requirement. It wasn’t until this final assignment that I noticed this deficiency. Again, in my previous journal I mentioned how setting up the tests and running them caused me to go back through and rewrite code quite a few times. So, I expected this final project to function correctly from the get-go. That wasn’t the case. The code ran well, but I noticed there were a few items that were not covered. So, on the surface, the code worked; however, there was no way of truly knowing if everything was covered. If I were to launch, there could have been some inputs that were errors that I would not have seen. I realized, this was just one more iteration. So, I did the same thing I had done in previous weeks with each java file. I implemented additional lines of code. One of the more notable examples was lines 48 through 50 on the ContactService.java file.

public List<Contact> getContacts() {  
 return contacts;

}

I hadn’t yet coded the return for the contacts array. I think this perfectly exemplifies the importance of a testing approach. Without running these tests, I wouldn’t have had this key element. Getting access to this array would be vital for implementation in future programming. This wasn’t the only change I had to make and so, I continued to go back through and rewrite my code several times, slowly turning red lines to green.

I know 80% coverage was the minimum and I ended with 82.5%; however, I do believe this is sufficient coverage for this application. I didn’t want to simply meet the requirements and move on. I wanted to know how and why certain lines were green, yellow, or even red. I still have several lines of code which are red under the coverage. I am ok with this. One example is line 30 in the ContactTest.java file.

@Test

void testContactIDNull() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact(null, "John", "Smith", "123456789", "123 Main St, City, ST 12345");

});

}

This line of code ran successfully, in that it gave me an Exception Error. That was the intended purpose. I could take it out to remove the red line, but then all branches wouldn’t be covered in the Contact.java file. So, while it brought down my coverage, the test was necessary. I was willing to accept this red line because I know it performed its purpose. It gave me the confidence that the actual functioning java file was correct. With this in mind, I went through and made sure all the functional files were covered. I have no red or even yellow lines in the functional files and all of my branches are covered. In the end, that is the whole point of testing, to ensure the code runs successfully with its intended purpose.

Experience

Two aspects of writing code are being technically sound and efficient. While a little out of order, I want to speak about efficiency first. When starting off this project, I wanted to stick to the basics. KISS comes to mind. This, while I am sure you are aware, stands for Keep It Simple Stupid. In personal dealings this can be difficult, but in design, its easier. With this as a guide, I read the requirements in Module 3-2 and used that as a basis to start writing my code. I watched the JUnit Tutorial video, as videos being included with requirements often convey a lot, I started there. I created the Contact.java file and filled it with the basic information. After that, I added the ContactTest.java file. With the first tests complete I created the ContactService.java file and then the ContactServiceTest.java file. It didn’t all work properly at first, and I had to add more lines of code. Each time I changed one file I had to revise another file. Yet, the entire time I only added enough code to make sure the requirements were met, and the tests were passed. If I had to add any additional information, I went out of my way to find the shortest method of writing said code. In fact, there are references to removing code to keep it as short as possible, often referred to as code quality. (Pressman, 2005) So, that is what I did in the final project. I followed this same method of approach and added only the necessary code to get the job done.

I spent some time describing my method of writing efficient code, but efficiency means nothing if the code doesn’t work. One thing I did was to try and watch, not just if my tests were passed, but also if they were covered. Manually reviewing my code, looking to make sure all the tests performed their intended function, was only the first step. I had to check that each of my branches were covered and all the lines in my functional Java files were highlighted in green. Double-checking these two and cross-referencing them is the best way to ensure all of the code functions.

# Reflection

Technique

One technique that has stood out to me is the technique of deciding what to test. I specifically manually reviewed my code, looking for full coverage of the functional .java files, specifically the Contact.java, ContactService.java, Task.java, TaskService.java, Appointment.java, and AppointmentService.java files. I made sure that these only had green lines, and I had diamonds next to each function denoting that all branches were covered. To do this, I had to write a few more tests in each of the testing Java files. The combination of getting complete coverage of these files along with successful testing results, in my opinion, is a perfect example of the technique of deciding what to test.

Another testing technique that I very recognizably used was testing early and testing often. When writing the program, I started off with the Contact.java file. I created the constructor using the five variables and then built the methods. I then built the ContactTest.java file. The point is that I created the first tests before my first file was even completed. Even after I had the Contact.java file and ContactTest.java file both completed and running in tandem, when I created the ContactService.java file and the ContactServiceTest.java file, I had to go back and rewrite the Contact.java file. Even with all four files complete, when integrating them together and running the coverage test, I had to rewrite some lines. Each time I added new lines I would run my tests and check the result. This technique of slowly adding lines of code and testing is very much the technique I employed as I built the project.

I was unable to use the testing technique of dividing my testing efforts. Although, this is very much something I would look forward to being able to do one day. I understand this project was used to measure quite a few metrics of us as students. It is used to test our individual understanding of Java, help introduce us to testing methods, as well as teach us software development techniques, and even a few more of which I may have missed the nuance. I know this class may not be the best time to introduce students to group projects, but I do look forward to the course where I get to collaborate with others.

The first technique I mentioned using was testing early and testing often. This is probably the most practical technique I have used on this project and one that I will continue to use in the future. An anecdotal story comes from another application I was building just a few months ago. I had one file that was calling a texture for a visualization. In total, I had about fifty textures. Based on the user’s selection, the desired texture would load. I had built this in two different menus, so much of my code was copy and paste each of the function calls from the other menu. I did not test it before packaging the project and loading it to an iPhone. I had quite a bit of other files in the application I was working on at the time, and it was a pure oversight on my part to not test the texture calls. Sure enough, I had typos, and the result did not function as intended. It wasn’t until after I opened the application on my iPhone that I recognized there was an issue with the texture calls. If I would have done a quick test on that before loading to the iPhone, I would have saved time repackaging the application and reloading it. This is something I now try to do often as I know how much time it can save, not always on writing code, but sometimes time on repackaging.

Choosing what to test is another area where I can see the value of implementing this testing technique, and the first step is to determine what test cases should be selected. (Cross, 2021) It wasn’t until the final project that I noticed the diamonds indicating that all branches were covered. This project was fairly small and choosing the areas that needed to be tested were straight forward. While I was able to cover all my branches and hit 100% coverage of all the functional .java files, I know this isn’t always the case. Getting a small project like this to start with is a valuable resource to be able to reflect on when building larger applications. I have worked on a few side projects that have been thousands of lines of code. Trying to write tests for every single function doesn’t seem feasible at that level. At a certain point, the concept of, minimum viable product, or MVP, comes to mind. Looking for the areas that see the highest traffic in the code and working my way down to lower importance methods and functions will be a skill that I am sure I will hone over the course of my career.

I do not want to sound like a broken record, but I am genuinely excited to work on a team. I know there have been countless times where I have written code and either seen other students code examples in discussion posts or searched through sites like Stack Overflow and seen discussions on similar programs to what I am working on. I am often amazed at how wildly they can vary. At the same time, I have seen two people write nearly identical code to perform the same thing without knowing each other. This diversity, and simultaneously, similarity is fascinating to me. Switching gears, I went into sales right out of high school and was forced to build my current career by building rapport with clients. The current industry that I am in is great, it just isn’t for me. I am a people person. I love to interact and converse with nearly everyone on anything and everything. My sales experience has taught me to not just be a people person, but how to communicate with people effectively. I know working on a team will be challenging, but that is why I am excited. I want to work with others for the challenge and to see how they work. I want to test my inter-personal skills on tackling a project with a team and see what crazy things WE can create.

Mindset

I have read a lot of articles on caution. In fact, I spent an entire evening trying to find out what that means. When searching “caution” along with “software testing,” there are countless articles with those words in the title, yet that same word seems to vanish from the authors lexicon when they form the structure of their work. Curious. It is up to me once again to try and convey the meaning I have taken from what it means to have caution in your code. At first, I wanted to say, I didn’t have caution, I wrote my code with reckless abandon, like some sort of hacker from a movie furiously typing away as green text poured down the screen. But in reality, it was often me staring at a blinking cursor trying to recall how to access that one piece of data in another function or how to insert that one piece of data into the right spot, inevitably resulting with me dropping my head on my desk and letting out a small groan of defeat. But as I read each article, and reread the rubric, the concept of caution slowly dawned on me. Caution is defined as, “Alertness and prudence in a hazardous situation; care; wariness.” (Dictionary.com, 2012) Alertness stood out to me. That awareness of potential danger. Obviously, there isn’t inherent physical danger in writing code, but it is there, nonetheless. If your code doesn’t function, as the author of your code, you are responsible. Your position, your job, or even your company could be subject to your errors. Caution is about taking the time to look for those errors. That is what writing tests is all about, trying to discover any errors in your code that will keep it from running with its intended purpose. That same caution you would feel if you lost your job, your livelihood, should be used every time you write a single line of code. That is what I did when I finished the project and closed Eclipse for the last time. I was confident that each of my tests covered what was necessary in the program and that it would run with its intended purpose.

Bias is interesting to me. We all have it, and yet, some people pretend they don’t. If anything, I think that is far more dangerous than openly admitting our biases and moving forward. But what does it mean to have bias in software? The very first thing that came to mind for me was my Turkish clients. I have somehow found myself endeared to quite a few clients from Turkey. Turkish names are quite different than to what I have been accustomed. My first thought was, would some of their names cause an error? I immediately replaced a few characters in my program and ran the tests again. Luckily, the names didn’t cause issues, but it was very much a wakeup call. I tried extremely hard to make sure I covered all the test cases only to realize, I don’t know what I don’t know. This is something I will absolutely have to think about going forward.

I am a bit of a perfectionist at times. I know, “perfectionist at times” is an oxymoron. I started a design company a few years ago. I had an original idea and knocked out the design relatively quickly. While I loved my product, I knew it could be better and made improvements to the design which ended up delaying production. Eventually, I lost external funding and had to sell off the company piecemeal for fractions of what I had invested of my own money. I was extremely young at the time and did not have great executive knowledge. That very expensive lesson taught me a valuable lesson about MVP. Being disciplined isn’t just about making sure you have a perfect product; it is about being able to make decisions. Once you have a functioning product, launch!

I am not advocating for a Theranos business model by any means. That is not what I am talking about. Your software must work. Period. For you to be confident in that, you must run tests, a lot of tests. I had nearly as many lines of testing code as I did for the actual program, and I don’t see that as a waste of time. Yet, I also don’t think it is necessary to get 100% code coverage. If the parts that you need are covered and you have gone through every possible iteration your imagination can think of, and it works, launch the product. These are two separate concepts that I think some people can get confused. I stress the fact that you need to make sure you have good code. I would try and boil it down to function versus feature. Your code must function, but there may come a point in time where you have to make the decision to put off a certain feature until version 1.1. You can always work on version 1.1 the second you hit the package button.

# Citations

Pressman, Roger S. (2005). Software Engineering: A Practitioner's Approach. McGraw-Hill Education. ISBN 0071267824.

Cross, Dr. Angel D. (2021) 6 Tips to Get started with Automation Testing. Southern New Hampshire University. <https://learn.snhu.edu/d2l/le/851372/discussions/posts/56577443/ViewAttachment?fileId=51651019>

Dictionary.com. (2012) Caution. Collins English Dictionary – Complete & Unabridged 2012 Digital Edition. <https://www.dictionary.com/browse/caution>