Alex Jarratt

Professor Maylon Walker

CS-320

Software Test Automation & QA

June 18th, 2022

Summary and Reflections Report

My approach to applying the unit tests across the modules remained fairly the same I felt. The requirements for each test in the software was more of a checklist than any actual testing being done. This then made the requirements the sole thing being tested with a success on the upholding of requirements for each assignment. I believe then that my approach was aligned exactly to the software requirements. For example, there was in Module Three’s Milestone Contact Class and Contact Service the requirements listed. There was a requirement that said each of Contact Class’s variable fields could not be null. Thus I formed a line for each test that checked for null before I checked for the variable’s validity. This would insure that each test would check if the variable was set to null at some point somehow. The first check for null came in at line 111 of ContactTest.java. The previous ones simply checked if there would be an existing item to compare to. I know my JUnit tests were effective based upon the overall percentage of my tests. My percentages were exceptionally high for getting used to JUnit at the time and I felt that I only improved since. Matching at least 80% coverage was quite nice to see and proves that my unit tests were effective at testing the code for issues. There are two instances where I know I made sure my code was technically sound and efficient. From these two instances I repeated this to my other files to keep consistency in my coding. The first instance is in my ContactServiceTest.java file. Immediately after my class declaration, I made a line I found incredibly efficient. I included each variable of the file in a single line as a protected String. While this did not assign a value to each, the declaration of each value here was important as I made a @BeforeEach call to make sure each test had the correct values it would need consistently. However, I cannot call values in a @BeforeEach (or at least to my knowledge this is not possible). Thus, my efficient line becomes all the more important as it fixed the declaration of variables that would be called for each test with the entered values in the @BeforeEach. This incidentally coincides with my technically sound instance as the @BeforeEach is a great way of making absolutely sure that each JUnit test gets the exact variables it needs to perform its test or tests. I used a combination of static and dynamic software testing techniques. My static testing mostly involved meeting requirement demands of each Module. This really means that my static testing consisted of looking at each modules requirements like in Module Four’s Milestone I had to make sure there was a character limit for each of the variables in the Task.java file. Additionally, this static testing would cover the not null value testing. For my dynamic testing I would tackle the fields that would change. Again in Module Four’s Milestone I had two dynamic fields because they were capable of being updated. Additionally, the Task Service file, which contained the code that would have been used for updating the two dynamic fields, also had to be able to modify the the amount of tasks that existed. This created two more dynamic fields that were a bit more complicated than the modifiable variables the ‘Name’ and ‘Description’ fields were. Thus, in one little file, I had at least four dynamic tests to get through in my initial look through. I did not use a non-functional test technique as there was no need to compare this to a industry standard. The bread and butter of testing comes from static and dynamic tests. I find that the majority of testing could be described as one or the other. Thus any project could use both of those tests, so long as they were tied to software development. It was not difficult at all to think that a check for a not null value would be nigh-universal in software development nor is it difficult to think that a check for if something can be changed when the code runs would also be just as necessary across most of software development. Non-functional however is not as necessary to a indie development group. Non-functional is mostly tied to making sure the code is up to par with industry standard practices and thus would be more useful in a major company in software development as they need to keep their code as cutting-edge as possible for sales. For this project, I had a fairly calm mindset. If I stress about a project of some fashion, I find that I do worse on it. Thus, I make sure that I stay calm and somewhat focused. If I find myself losing focus, I take a short break before coming back to my task at hand. Fairly straight forward I would hope. I employed caution by making sure I understood my code. Perhaps I don’t have a great understanding of it but I do have some idea of what goes on as the code runs. There was a moment where I was utterly lost with my own code whereupon I had the idea of taking my variable names, and changing them again. I still kept the naming scheme as that was part of the important coding practices that have long since been drilled into my head. However I changed one variable to the newer version, one at a time. I would take all instances of that variable in the entire file, and change it. If there was zero changes, I could make a theory out of what the variable did and test that by changing x, y, or z in the file. From there, I would continue this process for each of my assignments as it helped build an understanding of JUnit quickly. Each of my submissions and the project itself contain my final version of these variable with the others lost to time and memory loss. Thus, understanding brought caution. I understand I will always have some bias in my code. I try to eliminate it as best as I can, which helps by understanding my code, but bias is inevitable. Thus, it always becomes a concern if I was a software developer. From the JUnit tests I understand my code is thoroughly tested against itself and that if my JUnit tests are not of enough quality that they will bias towards clearing bad code as good. However, I also try and find code others in the past have done. This lets me see a potentially quality work that I can look at to see if I am missing some aspect of a file or a issue of some kind is apparent in my own that was rectified in the past by another person. I avoid outright copying their code, because that’s plagiarism, but also because I need to write it myself to truly help myself understand each assignment. I do not understand what I do not attempt after all. Being disciplined in my commitment to quality as a software engineering professional is all about what I do not accept for myself. I do not accept that simply looking over a past person’s work is all I need to do. I make sure I do honest attempts in all coding aspects as otherwise it feels like I’m losing out on a vital part of my future career. It is necessary to not cut corners when writing or testing code as this leads to bugs, errors and a lazy mindset. To simply deem a step not necessary out of hand is to admit to laziness. Laziness is inexcusable to any industry and I will not succumb to it. I plan on avoiding technical debt by always making sure each and every single one of my files is understood and wrote well. This can be achieved by my method of re-writing variable names that I mentioned earlier and through a variety of static and dynamic tests. This will ensure that what I produce will never be sloppy as I cannot accept sloppy work for myself.

References

Jarratt, Alex (2022). Project One [Computer Software]. Grand Strand Systems.

Module Three Milestone Guidelines and Rubric. (n.d.) Retrieved June 18, 2022 from

https://learn.snhu.edu/d2l/le/content/1069812/viewContent/18251037/ View

Module Four Milestone Guidelines and Rubric. (n.d.) Retrieved June 18, 2022 from

https://learn.snhu.edu/d2l/le/content/1069812/viewContent/18251040/ View