Jacob Mueller

6-18-2022

My approach to coding and developing my unit tests was mirroring the requirements as closely as possible in the tests. This required some close reading of the requirements for the variables and briefing documentation for the project’s parameters. If there was a limit on the length of an ID, I would apply an exception to cap the length of the ID variable, then I would have a test written to test that setup. In the case of the task class I tried to make sure that the ID couldn’t be changed after setting the ID, much like (I think) snapchat or gmail both of which lock in a user’s ID for a year minimum, and in order to meet that requirement I had the setter/mutator apply the ID to a final variable. I thought about implementing a counter that would stop at 1 change to the ID, which in hindsight seems like it would be easier to track and fix for front end customers now that I think about it, or at least something like it. For the appointment service I had to figure out how to work out the dates and got a little lost in that process, but it can be seen in the test that I do test for the requirement that the dates can’t have already passed. In the contact service class I was unable to figure out the ID calls, which shows up as a problem in the testing portion, which is in itself indicative of how closely I was trying to mirror the tests to the requirements.

I know that my Junit tests were effective, not only because I was twinning the base class and the tests but also because I would continually reference resources online to ensure that I wasn’t missing out on any other options. Looking at the implementation and escapement mishaps throughout the course resources and discussions I was aware that there were things that I wasn’t going to be aware of (Russia didn’t think clouds would be a problem for their scanners despite accounting for other airborne objects), looking at some of my tests I don’t like that I was getting lazy or having fun when I should have been getting more creative with my testing. I think I could have better coverage, I definitely took the “bare minimum” advice to a poor extreme, but I won’t be too hard on myself considering that I did make sure that the exceptions informed the tests I included.

My experience writing the Junit tests mirrors (sorry for the overuse) my experience with starting coding, I felt like I was learning an entirely new language but got the overall concept. Like, I was following along with some video lectures religiously, and didn’t really get too comfortable without that crutch on my own until the very end here. I got more comfortable with displaying and playing around with the concepts but don’t feel like I have enough of an understanding to weather an extended questioning on the usage.

I feel like I already addressed the efficiency and technical nature of my code with examples from the tests, but I will cover some of the finer points here as well:

The testing techniques used in this project and overall in the course were unit testing techniques. Integration testing is one of the techniques that comes to mind first because I like those anecdotes about how sometimes a final project will be ruined because half the plane was built differently from the other half. When a dev team is working on a project and the individual developers join their individual pieces of the project, the integration tests that have been run should mitigate any problems that arise from that joining. This kind of testing helps on a macro scale between developers, which is cool but doesn’t have the same learning curve that unit testing has in our current online course environment.

System and Acceptance testing are the other two testing technique options, system testing is the testing of the system near the end of the life cycle and acceptance testing tests the project against the customer’s requirements. System testing as previously stated occurs later on in the sdlc, that’s because it tests the entire system and all the integrated parts. Acceptance testing on the other hand doesn’t have to occur just toward the end of the sdlc since the customer could want updates at certain milestones, in much the same way that we (the class) have worked step by step on different blocks of code and tests.

My mindset was not good for this course, I recently started fostering with my wife and that put a lot of pressure on me, and I petulantly took that with me into the “classroom”. I made a few The Shining references in my tests that I thought were funny and helped me to express my feelings, but now that really sticks out to me as a poor choice. I wish I was more cautious and digging into my own code with the tests, because it isn’t on me to make the product important, it is my responsibility to take something important and ensure that it works correctly and safely.

I was very biased in my approach, as stated previously I think I phoned it in with my tests and the meeting of requirements. Hindsight being 20/20, I know that I and other developers need to leave any misgivings or pride at the door in order to maintain a better quality of code. I was quick to simplify the process by making variables with parameters matching the requirements and mirroring those with specific tests that only assessed the variables. I’m not the strongest coder, so, not taking my code seriously when I got to the testing portion of a project wouldn’t be digested in a work setting. It looks disrespectful if I look at it with a more discerning eye, I wouldn’t want to get a diagnostic report for my car or my shear at work where the tests were all the same and no extremes were tested at all, it would feel unsafe and confusing.

I’ve made my point by now, hamming up the tests and only barely touching on the requirements throughout the tests written won’t make anyone feel comfortable with their product. There’s a bit of advice I get from time to time that basically says, “thinking does nothing to change you, doing is what changes you”, basically, thinking about doing something new is nice but it doesn’t inform a new opinion, while actually taking steps to trying that new thing changes you. Thinking about how to complete a project is nice and can be helpful, but writing notes or pseudocode, looking through the resources, sitting down and coding, and talking to a rubber ducky will actually get the job done. Right now, I think I am in the process of nailing down the overall concept of Junit testing, looking back I can see that there is a little progress made but much more progress and refinement to be pursued. And that is what sticks out to me when thinking about the benefits of discipline in software development, putting the work in is what will benefit developers in the field, taking the licks now or early on in a career will be better than having always taken the easy way out being out of their depth.