Christian Drouin

Project 2

My first attempts at testing with Junit on the contact and contactService classes was not great. My contact class met all the requirements, but my strategy for the class was complicated and confusing. My contactService class was close to meeting requirements, but due to the complicated nature of the contact class and an incorrect package set up my contactService class was never functional. After receiving some guidance from the professor, I was able to accomplish all requirements for the task and taskService classes the following week.

My testing for the contact and the contactService class did not align with any of the software requirements. I was unsure what exactly should be tested which resulted in two assertion tests on my contact class with no further testing achieved. My task and taskService class met all the software requirements for testing. For the task class I validated that every field of a task instance would pass when given correct entries and fail when given incorrect entries. I also verified that you could create a task and update a task. For the taskService class I validated that you could add, update, and delete tasks. I ensured that for updating and deleting tasks the process could only be done using the tasks ID number. Many tests were done, and the classes certainly met their 80% test coverage standards.

I know that my overall Junit testing for the contact and contactService classes was a complete failure. I believe that my test coverage for task and taskService classes was extensive and was probably closer to 100% than the 80% minimum requirement. I unfortunately have been unable to perform a test coverage run. I have been trying to get my tests to show me the highlighted lines of coverage and the total test coverage that we have seen in our readings, but I have had no success.

Keeping the code technically sound was a matter of staying true to the requirements. I didn’t try to take shortcuts, and I didn’t try to complicate things. I wanted to make sure that the application needed was all that was delivered. To verify that what was created was safe to use I verified that in both task and taskService there was JUnit tests to verify that inputs should fail and succeed. I wanted a product that had been proven to succeed and proven to not fail. Unfortunately, as I have already mentioned the contact and contactService was not technically sound and would certainly not meet the customer’s needs.

To ensure that my code was efficient I heavily analyzed the code my teacher sent me for the previous classes. I used the same framework and methods as my teacher but re-worked them to fit the requirements for the task and taskService. Specifically line 9 of my task class creates a Boolean method which essentially takes a string item and an int length. This method is then used throughout the rest of the class to simplify other methods which kept the code extremely simple, maintainable, and secure. The code for my contact class was efficient in a few minor ways, but primarily it was disorganized and confusing.

The first two milestones I ran into many issues. The first milestone involved an extremely basic level of testing techniques that essentially revolved around just looking at the code I had created which could technically be categorized as static testing reviews. I did attempt to create some JUnit tests in this first milestone, but they were extremely limited and did not validate any of the classes and methods created. The second and third milestone employed dozens of JUnit tests. This JUnit testing is known as regression testing which fits underneath a larger form of testing known as dynamic testing. This regression testing can be repeated on both functional and non-functional requirements. The best part of regression testing is that it allows us to bypass many specific tests and quickly validate that our system is working within a certain level of accuracy. Regression testing fits beneath dynamic testing because it is used to test the dynamic behavior of the software at hand.

The two main testing techniques are static and dynamic testing. I briefly used static testing with the first milestone and heavily relied on dynamic testing for the second and third milestone. Each of these two testing techniques have many categories that fall underneath. For static testing there are several different types of reviews that can range from informal to highly technical and different types of analysis. Dynamic testing also has several subcategories including unit, integration, system, and acceptance testing. All these different styles of testing become applicable based on the application at hand and the level of testing that is required.

With all these different techniques it can quickly get overwhelming. All types of testing are useful, but all tests have a certain place as well. Is there a project out there that requires every single type of testing we have listed? Yes, it’s possible, but more than likely the choice in tests was broken down based on the requirements of the application and a few choices were made for both static and dynamic. Some projects may require highly intensive unit testing for important areas of the requirements documents, and other projects may require more system testing to validate the application as a whole. The reality is most projects will have a handful of many of these tests, but it is very expensive to test dynamically and the less testing you realistically need to do the better.

I employed the mindset of modularity for this project. Each class involved similar methods to achieve requirements and because of this I used the contact and contactService classes as a template for the other classes. It was vital that I properly developed the first two classes so that the other could be easily copy. I used extreme caution in the development of the later classes. I had already verified that the first two classes functioned as required, but I had to be careful replicating the code in other classes because although the requirements were similar, they were not identical. Many lines of code had to be changed from the rough template(contact and contactService classes) to match the requirements of the new classes, and I had to carefully review and verify that the new requirements had been met. For example, in the contact class there was five input variables with different length requirements. For the task class there was only three variables with their own set of length requirements. I tediously reviewed the code to verify that all areas of the task class had been properly changed from the original contact template.

It’s difficult to step away from bias when reviewing your own code. It is in our human nature to hide away our short comings and faults and it is not difficult to quickly dismiss coding malpractices and weak points just the same. To avoid this major issue, I had to remind myself that I am by no means perfect and that I am prone to errors. I had to go into my code with the idea that there is probably a better way to code the classes required. With this mindset I was able to look at my code not just for errors, but also for better solutions which kept my mind actively searching for better solutions and overall correctness. Bias is a huge problem when reviewing code, you helped or completely created. It is wise to avoid this predicament and have others review your code for you. One of the major issues I had that I am aware of, but couldn’t find a solution for, was the use of a Date() method inside the appointment classes. I carefully reviewed these sections of code to ensure that it worked and only looked at them so critically because I knew it could be done in a far better way.

Ultimately the most important area of testing seems to be discipline. As a software engineer it is our personal responsibility to develop secure, safe, and quality software products. To meet these responsibilities, we must test our code with extreme discipline to ensure any possible escaped code is not going to degrade our products in any capacity. Shortcutting testing is shortcutting a project because the reality is, if we don’t test it, we don’t know our own vulnerabilities. Without proper testing technical debt is a close guarantee, and we would much prefer to spend thousands on testing then millions on lawsuits, reworks, or lost clients. Look at Boeing. They didn’t properly test code for their autopilot software in the 737’s and because of it their planes malfunctioned and killed 300 plus people and landed them a 3-billion-dollar lawsuit. That’s the type of catastrophe no software engineer should encounter.