Ryan Resk

Southern New Hampshire University

My testing approach aligned with the software requirements by making sure that the program would not allow invalid or null inputs. These checks included length errors, empty checks, and other defensive programming techniques. For the appointments, there were checks for dates that were in the past, and invalid IDs and descriptions. For the contact service, I included tests for the first and last name, phone number, and address. Lastly, in the task service, I also had tests that covered invalid descriptions, names, and IDs. I know my JUnit tests were effective because my coverage percentage was greater than or equal to 80%. Although I did have an issue with my appointment class on line 51 for my id length check and my description length check on line 80.

I ensured my code was technically sound by following industry best standards. For example, in TaskTest.java (line 22 and beyond), I utilized as little lines of code as possible to create the tests as the more lines of code you have the more your chance of an error increases. I ensured my code was efficient through reducing my bloat as stated before, but also by utilizing the principle of polymorphism in order to reuse parts of my code for different tests, the best example of this is how I carried over my data layout from TaskTest.java (Line 12) and ContactTest.java (Line 14).

The software testing techniques that I employed in this project start with dynamic testing. The dynamic testing utilized in this project was primarily unit testing through JUnit, which divides the program into the smallest possible parts in order to test them against various scenarios. This is a form of defensive programming in which many inputs are checked against methods/functions in order to rule out either invalid or empty (null) inputs that could lead to unexpected run time results. The other software testing technique that helped me greatly to create efficient and technically sound code was static testing. Static testing consisted of me reviewing my documentation on the project, walking through the code, and technically reviewing it to find errors sometimes as simple as a syntax error. This form of testing is irreplaceable, as dynamic testing can only go so far if your program is not compiling or having run time errors leading to termination. If your program is terminating your dynamic testing will not work at all which is why I was led to static test. An extremely interesting and useful testing technique that I did not get to utilize during my work on this project is black box testing. Black box testing is a form of testing in which the tester does not have any working knowledge of the software at hand in order to most closely resemble the end user. This form is testing is absolutely priceless when it comes to UI heavy software, as the tester can navigate and utilize the software without understanding what every click does on the back end. This form of testing is sometimes the only way to detect those stubborn issues you are having trouble fixing as the developer and also the tester.

My mindset coming into this project was to create the most efficient, cleanest, technically sound code I could. I wanted to make something I could work around and make changes to without feeling like I am tangled in a web trying to plug holes. I wanted to understand what everything was doing and how it had an effect on run time. A specific example of this is the contact service, as I initially struggled quite a bit to understand how everything was working together. I took that knowledge of the relationships between the code, and I applied that to the task service and the project and it worked wonders for me in terms of truly knowing what I created. I tried to limit bias in my test coverage especially, as a developer and also the only tester, I have an innate bias to defensively program against things that I personally would do. This, however, is not good practice and it’s a bias that you simply cannot get away from, which is why any serious development is not developed and tested purely by one person with no help from software or other humans such as in black box testing. Discipline is in my opinion the most crucial aspect of programming and software development. Following industry best standards, creating technically sound and efficient code is very easy, until you have a program will hundreds or thousands of lines of code. All it takes is one night of working on a project where you slip up on your best practices and now you have a mess on your hands that you will have to work around for the entirety of the project. This project is really where I began to understand this principle of discipline and began implementing it into my toolbelt. An example of this from my work is best shown in my initial work on the contact service and how I was able to regroup and create a strong base to work toward the project with. Overall, I had a great experience with this class despite the difficulties and I have a much better understanding of discipline and the importance of doing things right the first time, even if it takes a little longer.