Tristin Raymond

CS320

CS320 Project 2

In terms of the JUnit testing that was done for this project, there were two things that I always kept in mind. The first was that I needed to make sure that the code was functional and performed the functions I asked it to. The second was that the code adhered to the requirements and the specifications designated by the client. Using JUnit tests validated both of these things and kept me on track to deliver a mobile application that the client would be satisfied with.

Almost all of the testing that was done was designed to be for the exact requirements that were asked of me by the client. Most of these requirements stated that certain fields cannot be null or exceed a certain number of characters. For instance, the description field of the Application.java file required that the description is not null and cannot exceed 50 characters.

*if (appointmentDescription == null || appointmentDescription.length() > 50) {return false;}*

To test this statement, I ran two JUnit tests. The first tested for a description that was over 50 characters to see if an error would be caught, and the second tested for a null description to see if the error would be caught.

*String description = null;*

*IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> {*

*new Appointment(ID, properDate, description);*

*});*

*assertEquals(“Invalid Description”, exception.getMessage());*

Based on coverage percentage, I know that my JUnit tests were at least 80% effective. Since I was testing using coverage, I could see where errors were in the code and what I needed to fix in order to make sure that the tests were actually covering what I was testing for.

To ensure that my code was technically sound, I relied on myself, the JUnit testing, and the Eclipse IDE that I was using. Between these three, I was able to check syntax, spelling, grammar, and even the functionality of the code.



Here I was testing the task of deleting a contact from the mobile app. To do this, I added 3 ‘dummy’ contacts to the test and then checked that there were 3 contacts. After this, I specified to delete contact “1”, which should reduce our list to 2 contacts. This was also confirmed in the test. The ‘for’ loop then runs through the list of contacts to make sure that the contact was actually deleted.

The main software testing techniques that I employed in this project were unit testing and static testing. The unit testing that I performed came in the form of JUnit testing and required me to physically write JUnit tests based on the requirements specified for the mobile application. The static testing came in the form of whitebox testing, where I manually took a look at the code for any bugs or errors that I might have introduced myself.

Since I was writing most of the program based on the requirements given to me, most of my code focused on the functionality of the mobile application. This meant that when I moved on to writing the JUnit tests, my mindset was aimed at writing tests that could discover any flaws in the requirements that I was writing the code for. There were more than a few times that I had to go back into my original code after running the JUnit tests to fix the logic of my programming.

One software testing technique that I would like to employ if I were to continue working on this project would be automated testing. Since I was running my JUnit coverage tests manually when I felt I was at a good stopping point, I was likely creating more errors as I was writing than I needed to. If I were running an automated build with the JUnit tests, I could have easily had it build the tests every time I finished a new line of code. This likely would have saved me time when going back in to fix any errors, as well as keep me from repeating mistakes in subsequent tests.

My mindset during this project went from confident to worried and then back to confident by the end. In a way, it was kind of a rollercoaster. Since this was my first time working with JUnit testing, I was definitely having a hard time with it at the beginning. However, towards the end of the project, I felt like I had finally started to get the hang of it, and I started viewing JUnit testing as more of an ally than an enemy. After seeing so many errors on the first coverage test that I ran, I began to employ much more caution towards the code I was writing. This not only improved my efficiency but also improved my focus towards the task at hand. One example of this was that I had duplicated variables on one of the files, which led to some mismatching when running the tests later on. I’m embarrassed to say it took me way longer to find and fix this problem than it should have.

When I am testing my own code, I definitely have a bias towards myself. I feel like I should innately know what sections of the code are perfect and what areas might be throwing an error. In reality, this is simply not the case. Because I have this bias, it tends to leave me more disheartened and frustrated when testing does not pan out the way I thought it might. If I were able to remove this bias from myself, then I would be able to accept my shortcomings more easily and grow faster as a programmer.

Discipline and commitment is of the utmost importance as a software engineer. It is very important not to cut corners in this industry as something so simple as a slight mistake or the absence of a unit can lead to catastrophic problems down the line. Software engineers should strive to avoid as much technical debt as possible. Technical debt is the future cost of an error that happens now due to the programmer choosing a quick and easy solution over a better or more efficient approach. The best way to avoid technical debt is simply by adhering to the best industry standard coding practices. This includes code reviews, the prioritization of code refactoring, and documenting all decisions and trade-offs throughout the development process. One example of this in the context of the mobile app being developed here could be the lack of a way to validate the ID of the contact that needs to be added, updated, or deleted. Without verifying that an ID already exists (or doesn’t), we cannot successfully rely on our code to perform the task we want it to perform.