**Project 2**

Ayden D. Hooke

Southern New Hampshire University

CS-320: Software Test Automation & QA

Dr. Bolton

2/27/2022

To begin, each piece of software that was written followed the requirements specified as closely as possible. In the first assignment, it was specified that the object should have a contact ID, a first name, a last name, a phone number, and an address that all needed to meet specific requirements. As instructed, all requirements for each variable were met. Each of these variables needed to be updatable and contacts needed to be able to be both added and deleted. Both of these things were accomplished. In the second assignment, there needed to be variables created with certain specifications, just as in the first assignment. Like the first assignment, each variable was created with its given restrictions. Also like the first assignment, objects needed to be able to be added to a list and they also needed to be able to be deleted, as well as most of them needing to be updatable. All of this was fulfilled, and it is all true for the third assignment as well.

With JUnit tests, you know they are effective if they have a high coverage. One can achieve a high coverage on their tests if they test each and every scenario. With all of the assignments, the JUnit coverage of my pieces of code was very high. This was due to meticulous testing to ensure that every case was covered so that each line of code would be able to be executed in the JUnit tests. It also was important to ensure that all of the conditions that I mentioned previously would be covered in the JUnit tests as well.

You know that your code is technically sound when it runs without errors and can successfully complete all the JUnit tests with a high coverage. Most importantly above all else, it is critical that you cover all exceptions that can be thrown by your code, as well as any other errors that might occur. By doing all of this, you can ensure that your code is technically sound. An example of catching errors is in the following lines of my code, where I catch illegal arguments (there are also many pieces of my code where I have catches for exceptions to ensure that no exception is left uncaught) :

if(taskID == null || taskID.length() > 10)

throw new IllegalArgumentException("Invalid task ID: Length over 10 or NULL");

You know that your code is efficient when it includes only the libraries that are necessary, doesn’t include variables and functions that are unused, and does things in a shorter way as opposed to a longer way. In a revision of my first assignment into what it is now, there are many places where I made my code much more efficient. One of these places is noticeable in the following lines of code that have made their way into every assignment of Project Two. This particular section of code replaced a significantly larger and significantly more convoluted section of my code from my original assignment:

private String generateTaskID() {

return UUID.randomUUID().toString().substring(0, Math.min(toString().length(), 10));

}

The testing technique employed in this project was the use of JUnit tests, which are used for testing the different parts of the data structures of a class. This includes the different variables introduced in the classes as well as the functions defined there. Specifically, with JUnit tests, you can test the different fail points for the data structures and functions. In the milestones from this class, there were many things that were tested. For example, in the first milestone, we were required to make a program that allowed contacts to be added, deleted, and updated. In the second milestone, we were required to do similar things to a list of tasks. The third milestone was also incredibly similar and required the same things to be done to a list of appointments instead, but we had to use slightly different methods in order to fulfill the criteria. Some of the things that were tested throughout the milestones were names or descriptions being too long, an ID being too long, anything being null, testing if a date had already passed, and some other things.

One software testing technique that I did not have the chance to use for the milestone is JBehave. JBehave tests the behavior of your code in different scenarios instead of JUnit tests which tests the architecture of your code like individual classes, functions, and variables.

Both of these tests are suitable for different environments and have their individual respective positives and negatives. JBehave is good for testing the behavior of your code and how it interacts with everything else, while JUnit tests are more of a static test that tests the structure and technical boundaries and capabilities of the code, rather than the performance and interactions of your code. JUnit testing fit the milestones’ testing the best because it was just static testing that was necessary to fulfill criteria provided. Through the use of JUnit tests, all of the criteria was completely met.

The mindset that I adopted while working on this project was one that would ensure that each piece of cound was sound, without errors, and would run smoothly in accordance with the specific instructions given for each selection of code. As a software tester, you have to employ caution such that every possibility presented in a piece of code is recognized and dealt with. No stone must be left unturned. This is something that you can directly check through the usage of JUnit tests. Since JUnit tests give a coverage amount that shows the amount of code that you have tested, you will always know whether or not you have tested every single possibility. You must use caution such that you fix even what might seem like just a harmless bug, because it may turn out to be detrimental.

As a software developer, it can be quite difficult to test your own code. After all, you are the one who wrote it. It is much more difficult to diagnose and test your own code than it is to test a stranger’s code. A developer is only seeing his work from a close-up perspective and might miss bugs that would be plain as day to see for anyone else since it is his own work. It is important that you limit your bias to your own code by using methods that will ensure there is no bias, such as JUnit testing instead of functional testing where you simply look over your code for errors and bugs. With my own code, I used JUnit tests to keep me in check, as well as the helpful tooltips given by Eclipse for when you do something wrong. By using these, I was able to see errors that I likely would not have caught on my own. An example of this is a coverage issue that I ran into. With some of the variables, I could have sworn that I had tested what would have happened after I had updated their values, however that was obviously not the case since the coverage on my JUnit tests was lower than where it should have been. It was there that I discovered my errors and then fixed them accordingly.

It is important to be disciplined in your commitment to quality code as a software engineering professional. Cutting corners is definitely not an option whatsoever when it comes to testing code. One little corner cut that someone might have thought was insignificant and unimportant can easily come back to haunt a developer later. Not only can it cause issues within the code itself, but sometimes it could introduce security flaws and other nasty things, even if it seemed like a “benign” bug. It is important to ensure that you test ALL of your code, no matter how tedious it is to write code to test all of that code. I plan to avoid technical debt by ensuring that every piece of code I write is written with the future in mind instead of just fixing an issue that I might have today. For example, in my code, I initially wanted to cut corners when it came to testing what happens when you try to create duplicate appointments. When I had done some functional testing, I had to do a double take because I realized I had caused an error that would end up deleting a customer’s data. After I noticed that, I made sure to go back and test every configuration of duplicates and slightly-not duplicates to ensure data integrity. Through that, I ensured that my code was error free and being disciplined had paid off.