Rico Applewhite

June 16, 2022

**7-2 Project Two**

**Summary:**

Through the JUnit test, I was able to run the test on each of the system’s requirements. Each of the tests had to be configured to contradict the unit’s specific requirement. An example from the contact class would be that the first name couldn’t be null or over ten characters. After writing the code, I tested it by creating an object that would have more than ten characters, in this case, "MichealAngelo” was used. The JUnit test was set to display the failed message "First Name has more than 10 characters." if it ran a check and the FirstName came back with more than 10 characters. If the test resulted had no errors or failure then it was a success. By doing this for every requirement, I’m assured that each one is met.

The same was done for the service classes. These tests tested every aspect of the code. My JUnit provides quality testing because it tested critical functions that the customers would often use. Unit tests for Add, Delete and each update was create. These tests was efficient at test nearly one hundred percent of the code.

A few ways I kept my code technically sound was by adding comments, these ensure that others who read my code will understand certain aspects of the code. An example of this is a line I used in the ContactService class, where some of the code was only there for testing purposes. This not only reminds me but lets others know that portion of the code can be removed after the testing phase. Ways I made the code efficient was by adding a way for the code to handle an input that doesn’t meet the requirements are handled. The majority of the requirement required a certain amount of characters. Using a boolean, allowed for any input over the required characters to be cut down to an acceptable amount. Using the example from early, when the JUnit ran the test with “MichealAngelo,” it didn’t fail because the output came back “MichealAng” which meets the requirement of 10 or fewer characters.

**Reflection**:

The primary Testing techniques that I used through all three modules were Specifications-based. The first technique is the equivalence partitioning technique where I used JUnit tests to test valid and invalid inputs. An example from my Contact and Contact test codes could be the character limits on the ID, first name, last name, phone number, and address. For each test, valid inputs were put in all areas except the field that I was testing. For example, while testing the requirement that the last name must not have more than 10 characters, I purposefully inputted "Micheal Angelo" which has thirteen characters. When the test is run it didn’t come back as failed because the way the code is written if Lastname has more than ten characters it is cut down to ten. So the input went in as "MichealAng"

The second technique I used was Use case-based testing which captures the interaction between user and system. This was used to test the services codes and involved adding, deleting, and updating information in an array. For Add, the user input would add a new Task to the array, and for deletion, the ID number the user input would delete the task from the array. Lastly, to update, the user inputs new information and the ID of the Task that they want to be updated. Techniques not used throughout the module include States Transition Testing, which tests outputs that are triggered by state changes and specific inputs. Decision table testing list all input and all possible output that can produce.

Although not used because they were not needed for this particular project these techniques still have their uses. States Transition Testing will need to be implemented in more complex systems with multiple states. For example, an ATM system will need to test states like withdraw, deposit, and balances. The decision table technique can be used in a system that determines qualification for something, for example, a Loan. The system will determine your eligible loan amount by credit score, months of employment, and age. Different inputs will cause different results that need to be tested.

During this project, I had to switch my mindset between focusing on coding and being a tester. “Testing is a technical professional with a significantly different mindset, and

with significantly different concepts and skills from those of the technical developer profession.” (Everett,2015) I had to employ more caution by not only making sure every aspect of the code followed the software requirements but that there were no holes in the code. An example could be the update feature in the contact class. I created the class so the contact had to follow the software requirements. This was extended to the update feature to prevent a user from bypassing the original requirements through updating. It is important to appreciate the complexity and interrelationships of the code because a single flaw in the code can create bigger problems for the overall software.

I believe limiting the bias of testing your code can be a tad difficult. As a developer, you know what you have written your code to do and bias can automatically arise. You could test what you intend the code to do and not test for what the user of the system could be able to do. Manipulation of the test could also be possible so that the code could fail the test, but the test could come back as a success. I tried to limit my own bias through the Junit tests, testing different inputs that the users may try that also contradict the software requirements. This is also where discipline comes into play. As I mention above, manipulating the test is possible if one’s goal is only to create a successful test rather than for the code to be tested successfully.“Billions of dollars in business are lost annually because companies and software vendors fail to adequately test their software systems and products.”(Everett, 2015) Cutting corner could cost your company not only money but their reputation and the trust of current and potential future clients.

**References**

Everett, G. D., & McLeod, R. (2015). Software testing : testing across the entire software development life cycle. IEEE Press.