Running head: Summary and Reflection

# Project Two: Summary and Reflection

# Alexis Alexander

# Southern New Hampshire University

# 

# Summary and Reflection

# Summary

# The approach that I took with testing software requirements was to establish the requirements outlined within the assignment. This was to validate that adding, deleting, and updating an object would work within the constraints of each variable. The way that I established this was via Junit testing with an assertion. This basically lets us know if requirements have been met with a true and false value.

# For example, in both assignments, the test was to return a true value if the object was added to the array within the task or contact service. Using this method was the best way for me to know that my unit testing had a wide array of coverage. Within the testing classes, I was able to input multiple test cases to ensure that the code was effective in catching any edge cases. The test cases also allowed me to input data that I knew would not pass the tests to make sure that the basic requirements were being met. A feature that I included in my task service assignment that I did not include in my contact service was the validation of variable constraints within the task class. This class consisted of my getters and setters for each input variable and the constraint addition further ensured that requirements were being met within the task service.

I attempted to create more efficient code by using array operations within both assignments. For example, to delete an item, I used the .remove operation in “**if**(taskList.getTaskId().equals(taskId)) {*tasks*.remove(taskList);”. To add an item, I used the .add operation in “**if**(!taskList.equals(task)) {*tasks*.add(task);”. I made sure that my code was technically sound by testing code coverage. This would make sure all possible cases of input and output would be validated and tested using my test cases. In the beginning, this was difficult as I couldn’t get a code coverage percentage over 40%. I learned through my test classes and the service classes that there was an issue with my boolean methods. When assigning a boolean to true within the method in a loop, I needed to insert a break statement to end the loop. Otherwise, it would be an infinite loop with an untrue boolean value.

# Reflection

Within each milestone, I have tested using white box testing. The form of white box testing that I employed for each milestone was the Junit test. This type of testing is used to verify the functionality of the code to ensure that the correct behavior is being performed. The Junit testing method was specifically used to test each method within each service. Within each test class, an assert method was used to have the test return true or false based on the user input. By performing the Junit testing, we were able to determine code coverage which helps us understand how much of the code was tested. A high percentage of code coverage and passing test cases helped me verify the functionality of the code that was written. This type of testing would be used after the review and again is used to test functionality. For example, if you had code that was supposed to produce a specific output, but you weren’t getting the correct results, you could perform this test to try to pinpoint the exact defect.

A technique that I did not use was the static analysis. This looks for defects in the code without executing the code. Once the code has been written, the static analysis is conducted via data flows and control flow graphs. The preference of testing the code before executing the code allows for early detection of defects that can sometimes be easier to fix early on. Another benefit from doing a static analysis is that it can provide early warning signs regarding code design. It could be that your code is just too complex, and the static analysis can warn you early on that your code is an open field for defects because it is too complex. This type of testing could be used if you just couldn’t get your Junit test to produce the correct output. For example, if you have a Junit test to run and your test cases are not passing, you could use a static analysis to ensure that the defect isn’t within some elaborate piece of code that you wrote.

The mindset that I adopted while testing code within the modules was very complex. I had to take quite a bit of caution after writing out the code for the services in correlation to the test classes. Initially, my test classes did not cover all edge cases and my initial response was to drastically change lines of code to make it work. Through doing this, I learned that I needed to closely inspect the lines of code within my service classes to figure out why the results of my test cases were incorrect. In doing this, I also had to eliminate a bit of bias towards my code. As I developed and tested the code myself, I was adamant that everything was correct despite the results of the test cases. I had to test as if I hadn’t created the code myself and found that it was a small error that was causing the failing tests.

Going through the process of software testing has taught me that discipline is necessary. I cut corners in writing my test cases in the first three modules as I was sure that my code worked. It wasn’t until I got to Project One that I realized that it may have worked but all the code wasn’t being tested. In my experience, it isn’t a big deal as this code will not be used for a real-world project. However, when testing code to be used in the real world, cutting a corner can mean that the user is missing out on a very important function that will be needed because your code doesn’t know how to handle the input in a method that you created.