Byron Laferriere

Dr. Cross

CS-320

Summary and Reflections Report

**Contents**

[Summary 3](#_Toc69651402)

[Testing Approach 3](#_Toc69651403)

[JUnit Test Writing Experience 4](#_Toc69651404)

[Reflection 5](#_Toc69651405)

[Testing Techniques 5](#_Toc69651406)

[Other Software Testing Techniques 5](#_Toc69651407)

[Practical Uses and Implications 6](#_Toc69651408)

[Mindset 7](#_Toc69651409)

[Caution 7](#_Toc69651410)

[Bias 7](#_Toc69651411)

[Discipline 8](#_Toc69651412)

[References: 9](#_Toc69651413)

# Summary

## Testing Approach

When viewing the project in hindsight, it was easy to pick out flaws that I did not like about my approach. I had to rework many of the tests due to an inconsistency in logic from java file to java file. My tests were originally missing 100% of the instructions supplied in the Contact Service file, due to this issue. The coverage tests were helpful in discovering why this was so. After running the coverage test finally, I saw where the test was beginning to fail and could trouble shoot from there. From this point, approaching the Task Service project, I kept consistency in mind and began focusing on how make the project easier for myself. Variable consistency in the Task Service and Appointment Service project made it easier for me to write the code and conceptualize how the program would work. I think that my approach to the requirements that were given to us was aligned, however, I now see how I could have written more tests to really push the software to fail. We learned about two styles of testing in the books, white-box and black-box testing, which I only used one of. I was attempting to fail the program according to black-box testing techniques and never really began to create tests that would test the internal structure of the program. Although the overall quality of the tests was verified through coverage testing above 80%, the tests that were performed were not extensive enough to reveal the need for more white-box techniques to be applied. The JUnit tests that were performed on the software were looking for the requirements documentation to be satisfied. The tests checked for the program’s response to null entries or entries that contained too many characters for the specified field, which was provided in this documentation. I know that my JUnit tests were extensive in this matter, because of the 92.9 coverage percentage that was achieved upon testing the program against the requirements.

## JUnit Test Writing Experience

For a beginning programmer, this section is always the most difficult for me to describe techniques used to ensure efficiency. My learning process thus far, has involved backtracking and reworking code after realizing that I had deviated along the way. After writing out the Contact Service class I realized that I needed to begin drafting out notes to follow when writing code to help ensure that it would remain technically sound. This project was a great first experience involving keeping track of variables used throughout multiple files and ensuring everything matched up. Using a sheet with pseudocode helped me with the Task Service and Appointment Service and protected me from having to retrace my steps because I was testing for something different than the parameters described in their class files. This can be seen in the segment of code in my TaskServiceTest file that used the code ‘public boolean validateID(String id) { if (id != null && id.length() <= 10) return true;’ to test that the entered value met the parameters in the class definitions. This is also a good example of code efficiency throughout my program files. By ensuring that testing was performed against the parameters and functionality requirements, I was able to increase my coverage percentage satisfaction with my JUnit tests. A good example of efficiency in my code was; ‘public Task (String taskId, String name, String description) {if (taskId == null || taskId.length() > 10) {throw new IllegalArgumentException("Invalid Task ID Entry"); .’ In this example I combined what could have been multiple if() loops for each parameter, to make my code more concise and efficient when iterating through. Using the or (||) syntax allowed me to combine two separate statements into one function.

# Reflection

## Testing Techniques

I did not realize how jaded the tests that I was writing were, until I began reading back through testing technique definitions. Black-box testing is described as tests that check the program against the requirements specifications (Hambling et al, 2015) or specification testing. From my experience, this has been a good place to start but it should not be considered as completely fulfilling testing needs. I used a combination of jUnit5 tests and Coverage testing to complete the project. Coverage testing is done over jUnit tests and provides a quantitative assessment of the extent and quality of the testing performed (Hambling et al, 2015). Performing these tests on the code that was written, provided me with a better understanding of how well the tests performed on the program repetitively. The purpose for doing this, is to ensure that if someone were to go in and edit the code, the tests would be able to catch errors or mistakes continuously in an autonomous manner. This is done by ensuring that the coverage testing performs well and results are as close to 100 as possible. Examples of me using black-box methodology can be seen in the classes designed for all three projects. The programs are written around specifications that were provided for us and never focused on how well the system would behave.

## Other Software Testing Techniques

White-Box testing is a good example of some other techniques that were not used or were used less than the ones previously described. White-Box testing are tests that are performed against the internal structure rather than the external functions of the system (Hambling et al, 2015). I found it interesting how the book mentioned that beginning programmers struggle with understanding the importance of separating what a system should do (specifications based) and how the system actually works (behavioral/design). White-Box testing uses different types of tests like statement coverages testing that are aimed at exercising programming statements (Hambling et al, 2015). There is also another form of testing used in white-box methodology called decision coverage testing. This test is used to measure the system performance at each decision point in the program, by dividing the decision outcome (Hambling et al, 2015). Obviously, techniques like Experience Based Techniques (Error Guessing/Exploratory Testing) are moot for my level of programming experience now, so these techniques were not used either.

## Practical Uses and Implications

The importance of using multiple types of techniques, while performing testing, will be most evident in the results produced by the tests. I can see how it is important to begin learning how to combine both types of testing when working on projects. Not only is important to keep specification-based modeling in mind but it is also important to consider how the system will perform while doing so. Depending on what type of project was being worked on, the software developer will need to choose which tests would produce the best coverage results for each case. This requires a software developer to understand how to create different types of tests, while using multiple techniques to do so. Being able to harness abilities like Error Guessing or Exploratory Testing, because of a time crunch, will help me develop into a better programmer. I have begun to learn the basics of black-box testing and its associated techniques like equivalence partitioning or use case testing but could also still further my knowledge in this area as well. The better I become at testing, the more beneficial I will be to a development team.

# Mindset

## Caution

When assessing my mindset that was adopted to complete this project, I can clearly see a vast growth in my knowledge and practiced techniques. At the beginning of the semester, I was unsure what the difference between testing and debugging was and now I feel like I have a solid foundation to continue learning and adding to. With that being said, caution was not even on my radar until we began doing the readings on the potential damage that could be done by incomplete software testing. As a software tester it is important to always keep caution in mind and proceed throughout projects once we are confident that all possible tests have been covered. This can help save countless dollars for companies and potentially even the lives of those depending on the software to function correctly. Complexity and interrelationships within the code we wrote was necessary to avoid redundancy. As a software tester, this is also an important practice to keep in our repertoires. However, once complexity begins to rise within the code, a need for continuous testing does as well. This is where automated testing needs to be incorporated into a project, which will help create test suites that are suitable for the code.

## Bias

Bias has always been an interesting topic to me, when referring to software development and how programmers inadvertently supply their own conditional biases into code. Obviously, I will defend my code when reviewing it, but it is an important practice to step back and view things from an alternative perspective. When I first began testing, this was a necessary practice to observe. I could not understand why my tests were covering such a little percentage at first and struggled with understanding how to enhance the results. I tried viewing the code through someone else’s eyes and asked myself the questions like ‘what purpose does this serve’ throughout. This helped see past my bias and once this was done, I was able to actually begin improving the quality of my coverage percentages by fixing unnecessary segments. This will be important to keep in mind as I continue progressing into this line of work. I’m sure that I will be responsible for testing my own code down the road, and at that point I will need to bring an unbiased opinion to the table to succeed. Remembering to question yourself as if someone else were there watching you write the code seemed to help me and I’m sure there are many more techniques that can help developers succeed at testing their own code.

## Discipline

Lastly, this course was a valuable reminder at practicing discipline continuously while writing code and tests. During the first piece of the project, I found myself switching variables and reworking through the java files over and over. After this I went back to the basics and wrote up pseudocode to follow as I wrote code. This allowed me to actually write code quicker than I had without spending the time to draft out a plan. The lesson learned here was that cutting corners will never help me out in the long run. Spending the time to draft up a plan of attack, actually created much more consistency and ease of configuration when writing the tests also. I plan to maintain this strategy when I begin my career in the field. It seems that this also helps me visualize the project better, which allows my creativity to flow better. This practice will help me avoid falling behind on my work as well, by ensuring that I am not wasting valuable time reworking projects or because of a lack of vision on where to take the project. In conclusion, the software testing fundamentals covered in this course will be combined with everything else we have covered thus far, to help me succeed as a software engineer.

# References:

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). Software testing - An ISTQB-BCS CERTIFIED Tester Foundation guide 4th edition. Retrieved March 04, 2021, from <https://learning.oreilly.com/library/view/software-testing-/9781780174921/08_chapter01.xhtml#h17>