# CS - 320

Summary and Reflections Report

David France

david.france@snhu.edu

Southern New Hampshire University

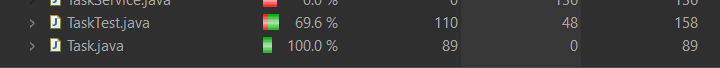
**Summary**

My unit testing approach was basically the same for each of the three features. Each feature has an object and a service to handle that object. I think my testing approach lined up with the software requirements for each class. Within the objects – Appointment, Contact, and Task – the requirements included various fields that couldn’t be null and were restricted in length. I created my tests to ensure that each of those fields would pass with valid input and fail with invalid input. This included testing passing and failing cases for both constructors and setters.

For the service classes – AppointmentService, ContactService, and TaskService – there were requirements for adding, removing, and updating objects. I created passing and failing test cases for each of these scenarios. I also handled assigning unique ids to each object within the service classes, so I tested the adding of multiple objects to ensure the ids were unique and incremented properly. Finally, I included tests for the service classes to handle null pointer and null HashMap cases.

I think my JUnit tests were very effective. I did a static review of each class to make sure all requirements were met, as well as a review of my tests to ensure all requirements were being tested for. Additionally, all my tests achieved 100% coverage for their associated classes:





Within the JUnit tests, I did a few things to make sure my code was technically sound. This included writing individual tests for each condition to make it easier to track failing tests. It also included testing boundary cases. I organized my tests into success and failing conditions, as well as constructor and setter cases.

I ensured my code was efficient by striving for 100% coverage without repeating code. One way I was able to accomplish this was by using the @BeforeEach tag to provide the setup for each test just once, but have it repeat for every test in the class. I also did this by testing boundary input, rather than choosing values in the middle of each partition. I employed the same strategy withing the classes to be tested by including the methods checkString(), checkPhone(), and isNull() to validate input and to guard against null pointers.

**Reflection**

For this project, I used a few static and dynamic software testing techniques. I used the static technique of reviewing the requirements and my code. I then created test conditions for dynamic testing based off those requirements. The test conditions all involved identifying test parameters, partitioning based off those, and identifying boundary cases. For example, one requirement was to include a description that was limited to 50 characters. I wrote a description that was exactly 50 characters for the success case and added one character for the failing case. This technique allowed me to both partition and boundary test at the same time. The other requirements were similarly defined, so I used the same technique for all my tests. I was able to measure the decision coverage of my dynamic testing by running coverage tests in Eclipse. In each class I ensured I had 100% coverage for dynamic tests.

Beyond reviewing requirements and my code, I did not use any other static software testing techniques. The main examples of these are group review and analysis. The review process involves a team getting together to go over a piece of testing software, with many people providing input. Analysis is usually an automated process that inspects code for errors or defects such as security vulnerabilities. Static testing is typically used to find defects in requirements, design, deviation from standards, or security (Hambling, et al., 2019). In the case of my software testing, the requirements were very simple, and they were all solo projects. This meant that many static testing techniques either didn’t apply or would have been beyond the scope of each project.

Another testing technique I didn’t delve much into was test-driven design. This is where you write the tests first, then design the code to pass those tests. While I think this is probably a good strategy overall, I found it too easy in these projects to write the code, then write the tests after. I think as I gain more experience as a tester it will become easier for me to identify and write test cases ahead of time.

All the above techniques have their practical uses. The techniques I used for dynamic testing of input validation and manual review of requirements represent the baseline level of testing. I would assume every project has passing and failing conditions, as well as a specific set of requirements that need to be manually reviewed to ensure they’re covered. As projects become more complex, analysis and review become much more important. Having others analyze and review code and test suites is to software development as an editor and proofreader are to publishing a book. It would be foolish to create a large project without that step. These tests would be particularly important in projects where security or regulations are big considerations. It would be disastrous to miss an important regulation because there was no review process. I would like to think this happens for most every project in the real world, but companies have been known to cut costs.

In addition to analysis and review, there would also likely be more involved dynamic testing, such as state transition tests. A specific project these would be important in would be an application that sends notifications to a user when an event occurs, or something has changed.

In reflecting on my mindset while working on this project, my main goal was to ensure all the requirements for each class were met. I employed caution by going over the requirements multiple times before submitting a project. I took the feedback from each submission and ensured I tested for null pointer conditions with my HashMaps. I also added in a test to ensure the class could handle a case where no HashMap had been created. I tried to eliminate bias by developing my classes and tests separately, although this is notoriously difficult while acting as both developer and tester. I think a good exercise would have been to test someone else’s code, thus gaining better insight into the role of just the tester.

For discipline and avoiding technical debt, I tried to make my code as clean and understandable as possible. I did this by using clear naming conventions in both my classes and tests, as well as clear and concise annotations. Additionally, I took the time to write separate tests for each individual requirement. While I could have written one test and still reached 100% coverage, there would be no way of knowing which test caused a failure. By writing all the tests individually and giving them clear names, any failure would be easy to track down. I plan on taking this strategy into the field so it’s always clear what I am doing and easier to track defects.

# References

Hambling, B., Hambling, B., Morgan, P., Samaroo, A., Thomson, G., & Williams, P. (2019). *Software Testing : An ISTQB-BCS Certified Tester Foundation guide - 4th edition.* Swinden, UK: BCS Learning & Development.