**CS 320**

**Project Two: Code Reflection**

**Steven Nave**

**10/14/2025**

In preparing for this journal reflection, I was able to really analyze not just my code, but my testing approach and how well that addressed the requirements. I found that my Task and TaskService files and tests were much better conducted than those of the Contact and ContactService. I had to go back and adjust the Contact coverage from 62.2% to 100% by adding 14 tests to the ContactTest.java file. Additionally, ContactService was already close with 91.2%, not hitting the full 100% because of a private helper method that itself was covered by 70%. This milestone really solidified both the process and quality goals for testing and trying to ensure not only coverage but really seeing where your code develops little logical errors with cascading effects, and how testing can catch them where they otherwise would have slipped silently by.

Consequentially, it is somewhat humbling and encouraging to see the pitfalls with the code and the subsequent improvements in the Java program, and being able to meet the requirements, and test them thoroughly, which more often than not either reinforced or exposed requirements that I thought I had already met. For instance, when I was going through the Task requirements, I had validation checks for both null and excessively long inputs, but forgot to check if it was empty, which was necessary for explicitly “required” fields for the object. But, using the coverage as a guide was very helpful, especially when comparing it to the use cases outlining the requirements. This was not something I was following as closely as I should when developing test cases and somehow missed that. I went back and corrected the issues with Contact and ContactService and was able to get: 14/14 tests for the original Contact.java with 62% coverage to 24/24 tests passing for 100% coverage; 10/10 tests to pass for ContactService.java; 16/16 tests to pass for Task.java with 100% coverage, and 14/14 tests to pass for TaskService.java with 89.8% coverage – similarly lower because of a private helper method.

Through the trial-and-error nature of repeating unit testing, I was able to see firsthand the utility of testing early and often, and very much trying to break the code that I had written, test it for stress factors and simple failing edge cases. All along there were hidden logical errors that would not have been caught without testing, or without a fresh pair of eyes. Such as trying to test for the testSetLastNameSucess() but I had contact.setFirstName() method within the testing function and compared it to setFirstName. So that test was passing, but I didn’t have the code coverage over the lines it was supposed to. Otherwise, I’m not sure I would have noticed, actually.

The JUnit testing files were quite efficient, although there were many variations of slightly different lines such as testing similar Contact objects with different constructor arguments, there was the ability to use the JUnit “BeforeEach” to construct a solid validContact multiple times for various tests, and similarly with the Task object tests.

Looking to the AppointmentService code, I followed a very similar process and really only had to address the additionally complexity of using the Date objects for the appointments, although through testing I, similarly, found more bugs than anticipated, especially because of the way the object instances were created. This forced me to inspect the Date objects created for the Appointments and AppointmentService more closely, and handle my test cases better, to properly ensure the correct behaviors were taking place, and not that I was writing poor test cases.

After completing the three project milestones for the mobile Contact Service application and drafting the various testing suites related to the components, it is easy to see the benefits of a variety of testing techniques and approaches. All the tests conducted for all the milestones were unit tests and mostly fell under the umbrella of the black box testing techniques.

There are many similarities between the use case requirements and thus the corresponding tests that were drafted for these components. One form of the black box testing techniques that was frequently used was the equivalence partition testing (Hambling et al, 2019). For example, in Contact.java, Task.java, and Appointment.java, cases had to be checked for the length of the ID numbers for each object. This is also true of several other fields in each object, but for now we’ll focus on the ID attribute as they each have that one, and all needed to be no more than 10 characters long. The equivalence partitions were 0, 1-10 inclusive, and anything greater than 10. In the test suites, these were separate tests in which they were checked to throw an exception if empty Strings or larger than ten or otherwise asserted to be correct when within the proper range of characters.

Similar to the equivalence partition in this case is the boundary value analysis testing (Hambling et al, 2019). Checking for the length of characters right at 10 characters long is equivalent to checking for all of the Strings of lengths 1-10, so just checking at exactly 10, just over the boundary at 11 and none gives us what we need to see for validation within the object. In addition, the same boundary analysis was used for the Date component in the Appointment class, ensuring that the valid future appointment dates worked properly when compared to the current Date, and that any in the past or null values were rejected and threw exceptions.

Now, these testing techniques are appropriate for the stage and scope of the application being programmed currently, but there are other testing techniques not used that may be useful for a more fleshed out application, such as white box testing and integration testing. If there was a separation of client and server codebases, as well as a database to map to, it would be incredibly important to test the integration of these components to ensure proper performance (Hambling et al, 2019). Additionally, end-to-end testing would be required before full production deployment. White box testing, in this context, would be used to validate the internal workings of each use case function.

Throughout the entirety of the development process for the Contact Service Java program, it was imperative to not only meet the use cases in the first place for the customer requirements, but also to check the bias involved in developing your own code, then testing it for bugs. A large part of the mindset necessary to perform both aspects of the one role well involves ensuring all use cases are met, and then ruthlessly testing the code, actively searching for bugs as if you are at war with the code, and it is hard to do without taking a step back and making some space before attempting to do so. Combing through the code with tests to ensure proper coverage and execution requires the neutral objectivity of separation between the developer side of your code and the testing development. It helps to pretend someone else wrote when developing the tests, and then when you find issues, approach it with curiosity rather than frustration.

Finally, this entire process requires a practice of discipline, in which both of the roles are taken seriously, as the quality of the code and performance of a real-world product that affects actual people, requires the appropriate level of seriousness from the technical practitioners and experts involved. Real damage can be done when the gravity of the work is not adequately appreciated, and so testing must be thorough and complete.

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing :*  *An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.