Brighton Ulery

CS 320

Prof. Mason

August 16, 2025

Summary and Reflections Report

While developing tests for each of the subsystems developed for this project, I took each of the software’s requirements into consideration. I reviewed the requirements for each, noting such things as the type of variable the data needed to be stored in, the maximum length of the field, and whether the field could be updated. For example, the unique ID for each subsystem was to be immutable, not able to be updated. As such, I ensured that each implementation of the unique ID was preceded by the ‘final’ keyword, indicating that it could not be changed, as well as ensuring it was private and inaccessible other than a getter function. After making these determinations, I wrote the source code as well as appropriate JUnit tests that ensured these requirements were being met.

The JUnit tests I wrote covered requirements such as ensuring null values were not being assigned and that value lengths were not longer than the maximum allowed. For more nuanced variables, such as phone numbers, I also ensured that they were of appropriate length and contained valid characters. Overall, my JUnit test coverage resulted in 84.2% coverage. This is considered a good test coverage percentage; however, it left room for some improvement in the testing phase if this system were ever to be put into production, where additional edge cases may need to be considered.

When designing and programming these systems, I put extra emphasis on ensuring my code was technically sound. I would often review the system requirements to ensure that everything I wrote met the standards laid out. I also followed programming best practices, using proper naming conventions and encapsulation. For example, each of the systems required objects of their related data holder (contact, task, appointment), which I wrote as classes to ensure that each contained its own relevant data and accessors.

Efficiency is also a big part of programming. We as developers want our programs to run smoothly and speedily. I took this into consideration when writing the code for these three projects. For instance, I have functions that search through the object list to determine if an object already exists with a unique identifier that the user is trying to assign. Rather than continuing to loop through the list after a matching object is found, I have it break out of the loop by returning with a relevant error message.

To ensure the quality and completeness of my code I used both static and automated testing. As I program, I continuously pause to test various parts of the system I am working on to ensure completeness and accuracy. The automated unit testing provided by JUnit allowed me to catch things that I had missed throughout my manual tests. I was able to write test cases that covered functional tests. There were plenty of techniques that I did not incorporate into my testing phase, such as non-functional testing, usability testing, and performance testing.

Software testing techniques are an important step in the development of code. Static testing is important as it allows a developer to quickly locate bugs or errors in their program. Unit testing allows developers to thoroughly test each section of their program and can be easily automated, lessening the amount of time that would take if it were done manually. An extremely important testing technique is functional testing, as these tests cover the specific requirements outlined by the client. Non-functional testing is also important, especially when a system incorporates accessibility, security, or quality of life features. Similarly, usability testing tests the system to ensure it is easily usable and accessible by the user. Performance testing is critical in environments were memory is limited or speed of access is of utmost importance.

The relationships that each of my methods had with not only each other but with other associated parts of the system, such as the service objects, was important to consider when writing my test cases. Just because one single part of my system passed a test does not mean that the entire thing was functional. I had to consider every method and the effect it had on other objects in the system to ensure more complete functionality was achieved. For example: when testing the setter methods in my service objects, I found several methods that passed, however they were assigning incorrect data to their related data object in a variable unrelated to the specific test.

Caution is essential to programmers, especially when our systems work with sensitive user data. We must ensure that our programs are technically sound and secure to ensure user trust in our systems. A strong factor in practicing caution is overcoming any kind of bias we may have about our programming abilities, especially when we are writing our own test cases. While writing code for these systems, I would often look at it critically, always finding something that could be changed or improved to ensure better functionality. If we allow our bias to encroach on our testing, we may inadvertently cause inefficiencies or data leakages.

Likewise, if we cut corners when we are writing our programs and tests, we may save some time and energy in the short term, but as users experience bugs and critical errors through their usage, we will soon find that that we spend more time fixing things that could have been corrected earlier in the process. As such, going forward with my programming journey, I will incorporate thorough and effective software testing techniques to ensure that my code not only meets requirements, but does so effectively and securely.