My testing approach was heavily connected to the software requirements. When creating the unit tests for the Contact, Task, and Appointment service packages, I went through the requirements and added tests as I read them. I tested to ensure my code could produce the desired outcome from the responses as well as throw exceptions when the requirements weren’t met.

Because I created the tests by going through each of the requirements, my tests should have perfect coverage. The only drops in coverage that I experienced were due to intentional exception throws. I even added a scenario which wasn’t explicitly specified in the requirements: if the input was blank (not null). If there are areas that have gone untested, I’m not aware of them.

The tests helped show me areas of my code that didn’t function as intended. This was usually due to a small mistake in one of the member functions, such as setting the wrong character limit for the task ID in the Task constructor. The testing allowed me to quickly know what parts of my code worked and what needed fixing. This allowed me to easily ensure code functionality. As far as efficiency is concerned, the tests also show how long each assertion takes. My code (when the tests passed) completed very quickly with no dramatic areas of slowdown.

I mostly used specification-based testing, as I tested based on the functional requirements of the system. Some specific techniques I used are equivalence partitioning and boundary value analysis. Equivalence partitioning ensures that each input falls into distinct categories, and that each input in said categories are equal to one another. As my tests were pass/fail, I did equivalence partitioning by testing multiple passing and failing inputs. Boundary value tests are designed to ensure proper functionality when inputs are near the line between two outcomes, as this is an area where problems often arise. While I did not write tests explicitly with boundary testing in mind, some of my tests do involve boundary values. Lastly, I created a single use case test for each service. These use cases allow me to check that each area of code works together. Other than that, the structure-based testing techniques of statement and decision testing were done through automated analysis of code.

Other specification-based testing techniques that I did not use include decision table and state transition testing. A decision table is useful for verifying specific outcomes given a number of inputs. As each decision in my code only relies on a single input, a decision table is not very helpful. State transition testing is used for checking functionality during events that change the program’s state. As my program currently has only a single state, there are no transitions to test.

When acting as a tester, I tried to make sure that the code responded predictably to various inputs. I made sure to test the individual functions of the system as well as test proper functionality of the system as a whole. For example, I tested the individual functions of the Appointment class, and also tested the interactions between the Appointment Service and a list of Appointment objects. I did this because, although testing various systems at once makes it harder to narrow down problems, the end goal of the program is to have the entire system function together. So, it’s important to recognize and test the individual code pieces as well as their interactions with each other.

I review my code based on how well it meets the project’s functional requirements. This way, as the code either meets the requirements or it doesn’t, there is no room for bias. Once I know the requirements are met, then I review the code based on how easy it is to understand. I try to simplify things and/or comment in explanations. I did this in the service classes whenever I thought that the declarations being made did not have an obvious purpose. I also did this explicitly in the Junit tests to make it clear what was being tested. That being said, it is impossible to omit bias in this area as “ease of understanding” is a subjective measure. While I enjoy working as both the developer and tester, it does introduce its own bias problems. What I’m specifically worried about only testing for things that I have already resolved during development. Odds are, if there is a specific case I think should be tested, then I already accounted for it in the code. It may be prudent to always have some tests be done by an individual other than the developer to account for this bias.

A discipline for quality is valuable as a software engineer as it helps maintain efficiency. Not only in the efficiency of the final product, but also in the efficiency of creating the product. Cutting corners ends up wasting time as those corners will need to be made later, whether by you or someone else. In this regard, being lazy actually results in more work. My strategy for avoiding this is quite simple, I feel physically ill when looking at bad code, so I strive to write clean code. As long as I don’t try to submit work that actively makes me vomit, I’ll avoid technical debt.