In terms of how my code approaches aligned to the software requirements, I made sure to read all the requirements one-by-one and consider them as I was writing code. This would not be as feasible when working on larger-scale projects with possibly hundreds of requirements, as they would be too much to remember when writing components one-by-one. Specifically, when requirements state that I need to ensure that a variable is not null and cannot exceed, say, twenty characters, I write a method to validate those requirements from the ground up. This can be seen in all Appointment.java, Contact.java, and Task.java.

The quality of my JUnit tests is on-par with what is needed to complete without any confusion, and they are not convoluted. To point to some specific examples, see AppointmentTest.java, line 30. The method name is titled “testAppointmentIDWithMoreThanTenCharacters”. This is self-explanatory as to what the test will do, and the contents are easily readable to see that it will create an appointment with a couple of valid values and pass. The failure condition will never be met except in extreme circumstances where there are a billion appointments. The same failure condition applies to tasks and contacts because they use the same type of ID system.

While writing the JUnit tests, I ensured that the code was technically sound by following IDE recommendations as to what types and modifiers to use on variables. For example, anything that does not need changed can be marked as final. One example of this is “idGenerator”, as seen on line 16 of Appointment.java. It is also marked as static because there is no reason to have multiple instances of idGenerator—otherwise it will start over at 0 again and will cause ID conflicts. I used the same methods (IDE recommendations) to ensure that the code was efficient and could not run into any exceptions where a value would, for example, return null or something unexpected.

It is a little bit difficult for me to pinpoint which specific software testing techniques I applied because there are so many, but I believe boundary value analysis was used when dealing with string inputs like names and descriptions, as seen in all three of Appointment.java, Contact.java, and Task.java. By “boundary value analysis,” I mean ensuring that given inputs fall within size limits. If a string is above a specified limit, it will be truncated. Something that could be done better is truncating with “…” instead of cutting it off or upping the limit before something is truncated. Another form of testing I employed was check list-based testing, where all the aspects of the application fall within the checklist of requirements.

During development, I did NOT make use of other testing techniques like risk-based testing, where there are potential risks in using the application. I am not aware of any risks in my application because of the methods I employed and unit testing I have added. Given the simplicity of the application, I noticed that not a whole lot of testing techniques \*could\* be applied to development. This might be a good thing for the smaller scope of it, but for larger projects, it would be good to research more.

Regarding the mindset when developing the application, I think the easiest explanation for all my decisions was that this was a smaller project without a whole lot of risk. Being given a list of requirements to comply with made the process of development easier overall, as making certain decisions would be unnecessary. In acting as a software tester, I applied a small amount of caution as I felt like I was safer in the testing environment. If prompted, I would have profiled the code to make sure it fell within a reasonable amount of memory usage. As mentioned before, however, the smaller scale of the project did not prompt much concern.

Trying to identify bias in my own code is a little difficult as I would write code the way \*I\* think is correct, but it may have some unknown flaws, inconsistencies, or runtime errors that I thought would not have happened otherwise. For example, in AppointmentService.java, on line 17, I made use of a HashMap to keep track of appointments instead of using another type of Map. I am not aware of the memory implications of using something simpler, which leads to a bias towards HashMap, because it is all I know to use in this instance. Anyway, I tried limiting bias by following the advice of my IDE and making sure that code met the requirements. Bias is a concern as it pertains to testing because the developer might not account for some edge cases. For example, when parsing an integer, sometimes you could unintentionally parse a value like 0xFF as an integer when you’re only expecting an integer input, like 255. Or perhaps failing to catch values that are above or below a limit. Or perhaps values that are not even of the same type!

It is important to be disciplined when writing code because from a realistic perspective, nobody is perfect, but developers should stive to write the best code they can. This does not strictly mean the most efficient or prettiest code, but code that strikes a balance between the two. It is also important to accept criticism from other developers and communicate to improve code when necessary. It is important to not cut corners because there could be edge cases not covered in a specific way code is handled. For example, using a built-in method to perform an action (such as sanitization) might not cover sanitization for database entry—potentially leading to SQL injection. To me, avoiding technical debt as a practitioner in the field means following requirements, learning from your past experiences, and NOT cutting corners. I am still learning from my mistakes of cutting corners today, having written code in open-source software that does not cover specific edge cases.