**Summary**

I think my testing approach was aligned very closely to the software requirements. Basically, my process of writing the tests was to look at the rubric and look at the requirements, such as for the task service: add tasks with unique ID, delete tasks per task ID, update name and description task fields per task ID – along with multiple requirements for each of those from the specifics of the task class requirements. With each of those we know we’re likely to have multiple tests cases. For instance, the name field can’t be null or over 20 characters. We’re going to have a case testing a name of appropriate length, one that is over 20 characters, and one that is null to see if it functions as we expect in each case. We do this for each requirement and ended up with 17 test cases for TaskServiceTest and 11 for TaskTest (and similar numbers for Contact and Appointment units as well), which seems appropriate and covered each partition of each test case as demonstrated by the previous example. All of these tests for all milestones passed, so I can point to that as thorough meeting of the requirements; I can point to exactly where tests show how the program reacts to each item on the rubric.

The experience of writing JUnit tests was not *uncomfortable* at first, but it was definitely something that took a fair bit of thinking and effort on my part. I had to thoroughly consider which method (such as the right Assertions) to use, how to measure success/failure, etc. However, after doing a few of these – such as by the time we were working on Appointment and Appointment Service classes – it felt like it became relatively easy since there was a lot of similarities and I was getting more comfortable with the process. Other than selecting the most appropriate Assertions, the most important part was to be mindful of different aspects that needed tested, such as testing both the appropriate length **and** an example that was too long (or short, etc).

I think my overall Junit test quality was good – the TaskTest returned 83.33% coverage with each test case scoring 100%. I was unsure about where that discrepancy comes from then – were there cases that *should* have been included that were missed?\* I researched and saw many developers aim for 80-90% coverage (at least without use of other automated tools that help catch edge cases), rationalizing that the final 10% is often not worth the trade-off of time/effort/cost to achieve 100% coverage. While I’m not sure that is the case for me, I think the results are effectively thorough to ensure the task class functions as desired. Similarly, the TaskServiceTest showed 88.5% coverage with similar 100% coverage across each test. For the Contact milestone tests, ContactServiceTest showed 82.8% coverage with each test reporting 100%, and the ContactTest showed 92.7% coverage with each test reporting at 100%. Finally, both Tests for the Appointment and Appointment Service class were over at least 84% as well. While I’d prefer 100%, I am estimating these to be sufficiently thorough results, given that I was thorough when designing cases based on the requirements and that 80% was set as our benchmark for the rubrics – so I feel comfortable defending this quality based on those marks.

I relied on the readings we had over Junit testing and examples/demos we saw to use appropriate test cases with technically sound code. A big part of this for me is having correct setup/initialization, which I did in the Contact milestone with a @BeforeEach void setUp method that contained the necessary information to create contacts that fit desired program functions (firstNameTest <= 10 characters) and those that deliberately fell outside that, such as a first name (tooLongFirstName) with 16 characters.\*\* Similarly to technically sound code is efficient code, and the biggest part of efficient code for testing to me felt like a similar answer to the first prompt – careful examination of requirements and setup that reflects that. I can then carefully input test case data to run before the application which then tests for (hopefully) each eventuality. A specific example would be that same previous example of a firstName in the Contact class needing to be 10 or fewer characters. I don’t need to write a test for every number of characters, 7, 8, 9, 10, 11… and so on. I simply need to use comparison operators to ensure that appropriate partitions are set, which were less than/equal to 10 and greater than 10. Those two partitions then need tested, which is far more efficient than attempting to test every/many values and follows equivalence partitioning ideology for efficient code for testing.

**Reflection**

There are a handful of software testing techniques that I used for this project. The vast majority would be considered primarily white-box techniques, as explained by Hambling (2015, p. 84) as: “Those based on deriving test cases directly from the structure of a component or system…[or] structure-based.” Unit testing is a great example of this, and since that is primarily what we focused on it makes total sense. Does this unit/component act the way I expect it to when given correct input – and what about when given *incorrect* input? I think some elements of black-box testing were utilized even though it was more of a white-box scenario, as I tried to utilize elements of the five types of specification-based testing – such as boundary value analysis (if a field had to be ten or fewer characters testing 11 characters, etc) and equivalence partitioning (rather than testing every number of characters for aforementioned example, test one that is appropriately less than/equal to 10 and one that is outside that range) when deciding what values would fulfill or not fulfill our requirements.

As described by Hambling, (2015, p. 84) evidence-based testing are “those based on deriving test cases from the tester’s experience of similar systems and general experience of testing, known as experience-based techniques.” Since I had no prior experience testing prior to this project/class, this is naturally an area of techniques I did not employ. If I had, those would typically consist of strategies like guessing for common errors I had seen before or conducting exploratory tests. Though I touched on black-box testing or specification-based testing, I did not employ many of those techniques since that didn’t really fit the objective of what we were accomplishing with unit-testing. Other types of specification-based testing include decision table testing, where a table of logical decisions is made to help track how various decisions would be made and affect each other; I *possibly* could have used this for some of the decisions, but most of the decision/conditional usage for our test cases was fairly basic so there wasn’t a lot of need for this. For somewhat similar reasons, I also did not use state transition testing or use case testing – these are designed for the black-box scenario where more of the external behavior is examined rather than the testing of each component, so they did not fit well for our purposes here.

As I alluded to with the last point, specification-based testing is more behavioral and often works well with black-box models where input is given and you’re examining the output – is it what is expected from a decision for instance, with decision table testing. For use cases, this is often a great real-world scenario to test the ways the system might be used the most. So, this is likely to be seen during the next stages of our project with Grand Strand Systems as we perform system testing and acceptance testing to model whether the mobile application communicates with the backend and functions the way it is expected, etc. At different stages and for different products, each different type of software testing technique will have their uses; I almost think of it like knowing which tool to use at the right time so I used those that fit the contact, task, and appointment services and will use these others mentioned as needed.

I tried to keep caution at the forefront of my mind as I tested this code, for a variety of reasons. Working too quickly or making assumptions as a tester is a really easy way to accidentally make a mistake or not catch a mistake that was already made. This can lead to a huge variety of problems, so it is always best to lead with caution *even if* that means taking longer to finish a project (or unit of a project). Similarly, respect to the relationships in the code is also vital to avoid larger problems down the road. If an aspect of the code that is being tested has a problem, does any other code interact with that code? For example, several elements of the Appointment and Appointment Service classes are interlinked in a variety of ways. To update or delete an appointment we utilize a searchForAppointment method – if that bit of code has an issue, then it causes ripple effects that have ramifications on the delete/update portions of our code – we could test the deleteAppointment method and assume it is problematic, when really the underlying searchForAppointment method could be the issue if it’s not properly working. So, paying mind to relationships of code is extremely pivotal in testing.

I tried to *assume* bias existed and counter that by forcing myself to review each code numerous times, at different times, and by talking aloud (rubber duck style) as I reviewed the code. I felt this was extremely needed since I was reviewing and testing my own code. From a developer standpoint, it becomes *much* easier to miss your own mistakes because you’re essentially too close to the project. Similarly to writing, you can write/type something and read it back numerous times and assume it is fine but someone else might catch it on the first pass – you’re reading what you were *supposed* to write – what you *meant* to do. However, we can’t trust that we’re perfect and that we always did exactly what we meant to and we in fact have to assume the opposite. The talking aloud method really helps with that on reviewing code, as that is an easy way to realize that I missed a line of code that was needed, or that I have an error in my logic. One specific example from this project was in creation of the Appointment Service class – maybe due to fatigue (since I had written similar code earlier in the project), I forgot to include tests for all the various ways new appointments could be added (in terms of how many parameters were included). I even missed this on my first pass of review, but on a second round as I compared the class to the test class I spotted my mistake. Without assuming bias and thinking that I had perfectly written everything I very easily might have missed this.

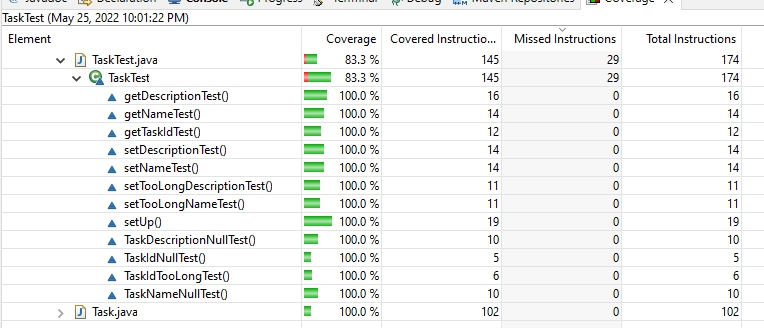
**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

Being disciplined in commitment to quality is an ethical question, even if not one of professionalism – it is both for me. I have a duty to the job and my organization/company to be diligent with quality code, as well as a duty to myself to continue to improve my skills each day. Past that, I have a duty as a software engineer to write error-free code (ACM Ethics).For our profession to continue to flourish, these types of standards must be met and cutting corners is a shortcut to damaging our profession. If the public cannot trust that we are writing code with attention to detail then it will dry up some opportunities.

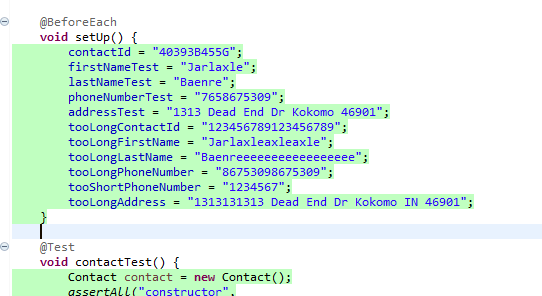
Additionally, the ounce of prevention is worth a pound of cure adage really comes to mind here. Cutting corners might save me a day here, an hour here, 15 minutes at a time once a week or so – but then the catastrophe from a huge error could result in loss of money for the company, disciplinary actions for myself (including loss of job and my own loss of money), even loss of life in extreme examples. So, a daily reminder – like a quote in my eyesight in my workspace is probably something I’d implement so that it’s always literally in my vision and then hopefully my brain as well to avoid cutting corners – it is not worth the cost! Refactoring code is another way to help avoid technical debt, as it can serve the purpose of review as well as cleaning up the code and ensuring that best practices were followed. Finally, I think keeping up to date both with best practices for the industry (and similar information like updates for the Code of Ethics, etc) is a strong step in prevention, as well as the flip-side of the token – I would periodically research *bad* examples of code gone wrong. This will serve as a reminder of what the ramifications can be to writing poor code and will serve as yet another cue that taking shortcuts with code will never be the right answer!

**Supporting Images**

\*TaskTest coverage



\*\* @BeforeEach void setUp() method showing technically sound code



**References**

ACM Ethics. (2021, October 22). *ACM Code of Ethics and Professional Conduct.* Retrieved June 16, 2022 from https://ethics.acm.org/

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software testing - An ISTQB-BCS certified tester foundation guide (3rd Edition) - 2.6 summary*. BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/pdf/id:kt00UC2I34/software-testing-an-istqb/life-cycle-summary