**Summary and reflections report**

My unit tests in project one were regarding the functionality of “Contact”, “Task” and “Appointment” tasks  
  
  
I ensured that my testing was in alignment with the software requirements by testing my code functionality to ensure the correct behavior of the various services we built. I also ensure software requirements alignment with extensive test coverage ranging from 80% in “Contact” to %90+ “Task” and “Appointment”. I know that these tests are effective because they are extensive as well as they cover all the main CRUD operations of our system.  
  
**Contact Service:**  
A screen shot of a computer code

Description automatically generatedFor example, I used junit to ensure aligned behaviour in CRUD operations. Attached is a snippet of that testing. First test checks for a normal addition of contact, while the second checks for an addition of contact while the array is full. Admittedly I used an array here, which is not the most efficient method as per feedback. Nonetheless, it is behaviour, not implementation that ultimately matters.  
  
**Task Service:**

A computer screen shot of a computer program

Description automatically generated

For example, Here I make use of advanced testing features like @BeforeEach and @AfterEach. This helps decrease any unintended testing behaviour by ensuring the tested variables are “wiped clean” before and after each test.

**Appointment Service:**

A computer screen shot of a program code

Description automatically generated

Here I use @before to ensure that my setup is correct before I begin testing. This snippet is a unit test that shows test setup and a test of a valid appointment test.  
  
  
**Experience:**

I had a good experience with unit testing. I guess my favourite part about unit testing is it gives you a sense of security while writing your code. This is more evident when you have to refactor your code as it helps to avoid code regression (decrease In quality/behavior of the code). I think my least favourite part was it sometimes seemed tedious to write so many unit tests, but I guess that’s the price to pay for high-quality code, which saves time and money, and for peace of mind.

1. **Technical Soundness:** We ensure technical soundness by checking for correct behaviour (output). For example, in the picture above, we can see a test for adding an appointment and then asserting to ensure matching output to the test parameters.
2. **Efficiency:** Above I attached a picture that clearly shows the use of testing features like @BeforeEach and @AfterEach. These testing features help in reducing redundant code by ensuring that the setup is correct before and after each test. Otherwise, we would have to create a new variable and instance in every single test.

**Reflections:**

* I used black box testing to test my code without caring about the implementation of the code being tested. This is good practice and its practical as this technique ensures we focus on **what the code does** (according to specifications) rather than on **how the code should work.** One negative implication of improper/omitting black-box testing is that the system we built is ultimately not up to the specifications, and thus the client would be unhappy and we have wasted our time and money. (Hambling, p. 101)
* I used White-box testing. White box testing focuses not on the output of the code per se, but that the particular elements of the structure itself are correctly executed. For example, testing if-else branches. (Hambling, p. 117). This is practical because it ensures every part of our code executes as planned and helps us catch logical errors. One negative implication of white-box testing missing or being improper is that because we didn’t the control flow of our program, our program might crash or not behave properly.

One example of code testing techniques I did not employ is “Experience-Based” testing Techniques for my code. The reason is that specifications were clearly defined, and I did not need testers to test my code so I could derive specifications and test-basis from them. This is obviously practical when we don’t have clear specifications to derive tests from. (Hambling, p.145)

**Note:** Learning about white-box testing, and black-box testing was very interesting because I studied cyber-security for a while, and learned about white-box hacking and black-box hacking. It is cool to see it from the other point of view, the point of view of a developer rather than a hacker.

**Minds:**

**Caution:** I guess my definition of caution would asking myself questions about the edge cases of my code (input, control flow) and taking test coverage seriously by writing unit tests for each scenario according to specs.

**Bias:** Bias is an interesting thing in testing. Because black-box testing's main purpose is basically this problem. It’s a way to test the actual functionality of the code according to specs without considering how it was implemented (how it should function). If I was responsible for testing my own code, then bias would be a problem because I would have the details of implementation in my mind rather than behaviour, however, I can mitigate that bias by using black-box testing. Ideally, I would have someone review my code and test it for me. For example, let's say I get used to testing a certain part of the code in a certain way, and for the sake of speed or just because of familiarity with the code, I might take shortcuts, or overlook important specs, or logical flaws.

**Discipline:** It is important to be disciplined of course. There are so many cases of code resulting in financial loss or worse, loss of life that it is incredibly irresponsible to not do our job of testing seriously just because it might be boring or tricky. Imagine if a civil engineer or a doctor took shortcuts because it was a bit boring, it would be nothing short of outrageous, and most likely punishable by law. Software engineering is a field of engineering even if sometimes we are referred to as “devs” or “programmers”. This is especially true when we cross a certain complexity threshold like working on medical devices that are the difference between life and death, or advanced computer simulations that cost a lot of money to run and develop. As for technical debt, I think communication and good management are very underrated in this regard. For example, having a product manager who can engage with stakeholders regarding reasonable deadlines. Having a team lead who ensures development is going smoothly and is able to take on responsibility to solve problems that are guaranteed to arise. Another good way to avoid technical debt is by simply doing unit testing and integration testing to prevent regression in code. Doing this would make refactoring much less risky and quicker while maintaining quality.