I think the extent to which my testing approached aligned to the software requirements were as close to exact as possible. I tried to follow the requirements one to one essentially. Like for example, in the TaskService one of the requirements was the task service shall be able to add tasks with a unique ID. This requirement is directly addressed by the testAddTask method in the JUnit test. It verifies that tasks can be successfully added to the TaskService, ensuring that each task has a unique ID. There are more examples of it matching up to what the requirements were. I saw the requirements and went step by step implementing them while also verifying them through testing.

I feel like the quality of my tests are good. I followed each of the requirements closely, my test coverage is good, and I focused on all the things I needed to do specifically. The coverage of each of my tests were at least above 60% some even 100% and those are marks of good coverage. Each test is made to check one thing at a time which helps catch any issues early on. I also made sure that every test includes checks to make sure the code does what it is supposed to. They ways in which the test are written would help me catch issues early so I could fix them if need be. A lot of effort was put into the quality of my test, and I think that it shows.

To ensure my code was technically sound I followed good practices. For instance, each method in the JUnit test focused on testing specific aspects of the requirements like for example in the TaskService class functionality, such as adding a task, deleting a task, updating task name, and updating task description. Those lines of code looked like: A computer code with numbers and letters

Description automatically generated Code like this ensures each functionality is tested and isolated, making it easier to identify issues.

This also ensures efficiency, to make sure I was efficient I focused on isolating each test so that each test method was independent and doesn’t need to rely on the other test. This means the test can be done in any order, which improves efficiency as well. By following some of these steps and more it ensures that the tests are technically sound and efficient.

During the software testing I mainly used two testing techniques. I did unit testing and integration testing. Unit testing is basically checking small parts of the software making sure they work right on their own, this was done using a tool JUnit. Integration testing is about making sure those parts work well together, this was also kind of done during JUnit testing. By doing both of these I made sure the code worked well and fit together smoothly and using that tool was of much help.

There are other testing techniques out there and I didn’t use all of them. One type of testing I didn’t use is performance testing. Another type of testing I did not use was security testing either. Performance testing would be like seeing how the software does under a specific kind of pressure or if used a certain type of way. Security testing would be testing how secure the software is and how safe it is to use. I did not do these types of testing, but they are important.

A practical use for performance testing would be seeing how well the software performs under certain conditions like high user loads, or heavy data processing. It helps identify if there are any performance issues. Performance testing ensures that the software would meet performance requirements.

A practical use of security testing is that it helps identify weaknesses such as unauthorized access, data breaches, or attacks, and ensures that sensitive data would remain protected. If there is poor security in software that comes with many risks, it should be one of the highest priorities of any type of software development. If there’s a security breech it impacts both the company and the clients and would make there not be good faith in the people who made the software.

While working on this project, I made sure to be careful and thorough, especially when testing the software. I understood that it's important to check how different parts of the code work together. For example, when testing tasks and their service, I tested each task feature carefully. In the tests, I checked things like what happens if you try to set a task's name to something too long. I also made sure to consider how changes in one part of the code might affect others. This careful approach helped me find and fix any problems in the software, making sure it works well.

I tried my best to stay fair and objective. I focused on following the rules and standards set for the project, rather than letting my personal opinions influence my judgments. For example, when checking the Task and TaskService classes, I made sure they met the specific requirements laid out, without letting my own preferences get in the way. If I had to test my own code as a developer, bias could be a problem. I might miss mistakes or problems because I'm too close to the code and like it too much. I might also ignore other ways of doing things because I'm stuck on my own ideas. To avoid this, I'd need to be open to criticism and use tools to check my code objectively. That way, I can make sure it's really good, even if I'm testing my own work.

Being disciplined in maintaining code quality is important because rushing through writing or testing code can lead to bugs and poor performance, harming the software's reliability and user experience. Avoiding technical debt, which occurs when shortcuts are taken during development, is crucial for long-term success. To prevent technical debt, I'll focus on code quality, do thorough testing, and regularly review and refactor code to make sure it remains maintainable.