The requirements for the software requisitioned by the customer included the ability to store and manage contacts, tasks, and appointments. Each of these required classes to generate objects related to them, and they required service classes to handle storing, reading, modifying, and deleting contacts, tasks, and appointments. Each of these objects had associated data to be stored along with them. The service classes contained functionality to modify some or all of their associated objects. JUnit tests were also required in order to automatically test that all of the customer requirements were met and that the program was robust.

The JUnit tests were effective in ensuring the product met or exceeded customer expectations due to the level of coverage included in the individual tests. More than 80% of the code was tested. The requirements themselves were tested using black box testing methods, but white box testing was also used to ensure that the inner workings of the program worked robustly as well. The JUnit tests went above and beyond to guarantee the highest level of quality possible within a reasonable time limit. The tests were made to be technically sound as well. This was accomplished by various methods including testing to see that illegal argument exceptions were thrown when incorrect arguments were provided to the classes during object generation or modification, as in line 45 through 54 of the TaskTest.java program. The tests were also made to be efficient by only using the resources necessary to adequately and fully check each tested requirement. An example of this can be found in the ContactServiceTest.java program. From lines 32 to 42, we are testing that the task service object can effectively delete contacts from the provided ArrayList. We only generate 3 contacts so that there are multiple objects in the data type but not enough to slow down the program or use unnecessary resources.

Due to the nature of the requested software, all of the tests that were developed for this program were either black box tests or white box tests. No non-functional tests were used to test this product. Black box testing is the process of testing the specific requirements laid out by the customer to ensure that the program is in line with their needs. None of the internal parts of the program are tested in black box testing. White box testing requires one to test the internal parts of the program. This type of testing ensures that the program is effective, efficient, and robust in its construction from the inside. Together, these two methods of testing provide a strong method to reach the maximum coverage necessary to ensure software quality. Non-functional requirements test more abstract details of the program including, but not limited to, performance, security, and ease of use. This program will not provide a user interface, is system agnostic, and is insufficient to display security. Therefore, non-functional testing was not used as it would not have been helpful to test these more wide-reaching aspects of a system. Black box testing and white box testing are necessary for any software project in which code is being written or used. Black box testing helps to show the customer that the product works in the way they requested. White box testing covers the areas that black box testing leaves out by testing the inner workings of the program. Non-functional testing is helpful in any system that provides a method for the collection of user input, a UI/UX, hardware, security, or accesses a database (Hambling et al., 2019).

In order to test effectively, I had to have a removed mindset. I had to consider that any part of the code could be broken, and that any of the ideas that I had may have been flawed. I had to be cautious by testing everything that I could reasonably test to ensure that the program was airtight. This program included many objects with a lot of data that could be stored, and there were a lot of interrelationships between parts of the system. Understanding this helped me to realize that there were a very large number of things that could go wrong. Some issues could even hide themselves until specifically looked for. I maintained this caution be even going as far as to test that the getters were working correctly as evidenced in AppointmentTest.java from lines 20 to 34.

Bias is a very easy thing to be influenced by when attempting to test one’s own code. It’s something that we will all have to learn to deal with due to the prevalence of and requirement to self-test our code. I worked hard to eliminate bias by separating myself from the code emotionally and treating it as if I was testing someone else’s code. Without emotional attachment, there was no ego, self-doubt, pride, or embarrassment to influence me to pass over things that should be tested. I also had to accept that, even with all of these measures, I could still miss something. Equipped with this understanding, I was able to write the best tests I could with the knowledge I currently have. It is also always worth noting that a fresh pair of eyes and ideas is one of the best ways to find and fix issues. If provided with the opportunity, I would always allow my code to be tested by others to ensure the highest quality.

Being disciplined in my commitment to quality as a software engineer is paramount. Many projects fail due to rushing, cutting corners, and making assumptions without fully testing and ensuring high quality. Failed deployments equate to lost money, lost time, and lost reputation. Sometimes fatal errors and bugs can render a program unfixable. Even if the software was fixed, it could be so damaging to the public’s trust that the new solution will not be accepted. Avoiding technical debt can be difficult due to the pressures involved in the ever-progressing world of software, but it is not impossible. I plan to avoid this as much as possible by continually studying to ensure that I have the most up to date knowledge in my field, planning and writing code with intention and thought, testing the code the fullest degree possible, and opening up the code to peer-review as often as possible.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Retrospective On Software Quality And Java Testing. In *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed., pp. 22–25). essay, BCS Learning and Development.