The testing approach aligned to the software requirements because it tested the required methods that were implemented in the base code. The first name and surname name couldn't be more than 10 characters long in the contact class, according to the regulations. In order to test for excessively lengthy input, the ContactTestJUnit created the assertion "Assertions.assertThrows (IllegalArgumentException.class, ()->new Contact("CID1029F847A6", "John", "Smith", "5553331234","1 Audrey Jersey City NJ 07305");". The job id could not exceed 10 characters, according to the task class specifications. As a result, the "void updateContactIdTest" function was added to the TaskTest JUnit to raise an invalid argument flag if the task id is longer than 10 characters.

JUnit tests are becoming better over time. The resources included in the modules were really helpful. In comparison to the tasktest, the coverage percentage was greater in the contact test. The service tests, however, exhibited a significantly greater coverage rate for both functions. I was able to gauge how well the functions were covered by the tests by the fact that they indicated a positive percentage. To make sure that each statement is used in at least one test case, I plan to utilize these more frequently.

I took a few steps to make sure the code was sound technically. I tried using data structures like arrays. I created a list for the strings in the contact class using "private static ListString> CONTACT_IDS = new ArrayListString>();". I made an effort to employ well used algorithms like equals, add, and length. Consider the following examples: "taskId.length()" in the taskclass; "getFirstName().equals("Jane"));" in the contact test JUnit; and "contactService.add(c1));" in the contact service JUnit. I also used the assertions "assertequals", "asserttrue", and "assertthrow". Because they were all structure- or specification-based, the software testing methods I used for the milestones would fall within the black box and white box categories.Black box approaches,

in the words of Knovel, "derive test cases directly from the specification or from some other kind of model of what the system should do" (Hambling et al., 2015). Equivalence partitioning, decisiontables, and state transition testing are examples of black box techniques that are used to test for both valid and incorrect inputs. the current state, outputs produced during testing, use cases created from test cases, and test boundary values. The coverage tests that investigate the components and the if-then statements made extensive use of structure-based testing. It is used to divide tests into testable chunks.Statement coverage, path coverage, and branch coverage are examples of approaches based on structures.Techniques for structure-based testing "are used to explore system or component structures at several levels" (Hambling et al., 2015).

The experience-based testing methods are the ones I did not employ for the milestones. These methods, in the words of Knovel, "use the users' and the testers' experience to determine the most important areas of a system and to exercise these areas in ways that are both consistent with expected use (and abuse) and likely to be the sites of errors - this is where the experience comes in" (Hambling et al., 2015). Exploratory testing and error guessing are some of the specialized strategies. Error guessing is the process of selecting the tests that will evaluate the code the most effectively based on past knowledge. Investigative testing is performed to test regions where there aren't enough requirements. Due to my poor testing expertise, I left these procedures out. Black-box techniques, white-box techniques, and experience-based approaches are some of the techniques mentioned above. Black-box approaches are typically utilized for outsourced testing when code portions have distinct functionalities. When the expected outcome of the product being evaluated is fully understood, white-box procedures are employed. Experience-based procedures are used "where specifications are either missing or inadequate and where there is severe time pressure" to "identify special tests that may not be easy to capture by the more

formal techniques' ' (Hambling et al., 2015). Based on their actual applications and ramifications for various software development projects and scenarios, each of these strategies is put into practice.

When working on this project, my attitude was analytical, exploratory, and growth-oriented. I exercised caution by conducting a lot of research, testing, and trial-and-error runs while watching tutorials. Given how much it affects the product's quality and performance, it is essential to understand the code's complexity and linkages. For instance, the coverage % was low when I first ran my coverage tests. The coverage percentage of my completed product, which added more tests and covered a larger portion of the code, exceeded the required 80%. The additional tests made sure that the code was sound technically and that the end result was of the highest caliber.

I tested everything repeatedly, regardless of whether I was certain it would function well or not, in an effort to reduce prejudice in my assessment of the code. We all know what occurs when we presume, therefore I attempted to develop hypotheses rather than assumptions. Therefore, if I were in charge of testing my own code, I could see how bias may be a problem. For instance, I may have missed the fact that the ID test wasn't working because of a missing line of code if I simply tested the function that verified that the length of the first name was no greater than 10 characters. Testing both valid and invalid input as opposed to simply one or the other also helps to reduce bias.

As a professional in software engineering, it is crucial to maintain and "advance the integrity and reputation of the profession" by a disciplined dedication to quality (Software Engineering Code, 2018). To prevent compromising the quality and performance of the completed product, it's critical to avoid taking short cuts. The software engineering code of ethics states that "software

engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest" (Software Engineering Code, 2018). This statement summarizes the duties of a software engineer. I intend to use agile development practices, which often test code, continually strive for high-quality, demonstrable software creation, and maintain open communication between the customer and the developers, in order to minimize technological debt as a practitioner in the area.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 4.1.1.1 The Test Development Process (K3). BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/pdf/id:kt00UC2IS4/software-testing-an-istqb/test-development-process

Software Engineering Code. ACM Ethics. (2018, December 19).https://ethics.acm.org/code-of-ethics/software-engineering-code/