In testing the software, I ensured the program I created aligned with the software requirements in a two-phase approach. First, I made a checklist of all features listed in the assignment requirements. I created this list considering both the rubric and the detailed assignment description. I broke each requirement down into to true/false statements. For example: “null task IDs throw an exception”, “task IDs longer than 10 characters throw an exception”, and “task IDs less than or equal to 10 characters are accepted” were the statements related to the requirement specific to the ID for tasks (for the task class, not for the task service class). Then, when building tests, I would make sure that each of these statements was true. Then, after creating the tests, I did a static review of the test code to make sure that the test was designed to truly validate what the requirements were looking for. For example, if I had a binary search tree with two contacts (IDs equal to 123 and 456), and I was testing removing a contact by deleting contact with ID 123, to validate the delete function, I could not simply check that the tree only contained one contact. I needed to verify that contact 123 was no longer in the tree.

JUnit tests were another critical tool for ensuring effectiveness of meeting the requirements. I wrote the tests and then modified them iteratively until I had 100% coverage on all of the java classes. Although 100% coverage may not be entirely necessary in all cases, this project was small enough that it was reasonable to reach the 100% coverage mark.

To ensure my code was technically sound, I read through the antipatterns listed in Chapter 6 of *Mastering Software Testing with JUnit 5,* published in 2017by Boni Garcia. I worked to understand why each of those antipatterns could be harmful so that I could write tests in a way that avoided it. The “Happy Path”, or design of test that only tests a single condition and “The liar”, or the design of a test that does not test what it is supposed to be testing, were the antipatterns that I had to really look out for. To avoid falling into each of these patterns, I had to think really hard about what the purpose of each method was for and design tests to ensure that purpose could be met regardless of the input. One way I addressed “the liar” was by creating a method that would create strings of a specified length. This helped me ensure my boundary tests were accurate. This method and its implementation is shown below.  
**private** String repeat(String character, **int** length) {  
 String outputString = **new** String(**new** **char**[length]).replace("\0", character);  
 **return** outputString;}  
@Test  
**void** testNameEqualToBoundary() {  
 String testString = repeat(testCharacter, nameBoundary);  
 Assertions.*assertAll*( () -> { **new** Task(id, testString, description);});

To ensure code was efficient, I used a BeforeEach setUp method that would set variables to pass tests. This way, I knew all tests would have the same standard. An additional benefit of this is if the boundaries ever change for one or all parameters, fewer modifications would be needed. A teardown method sets the variable values to null to prevent excessive memory usage. The setUp for TaskServiceTest is shown below:  
@BeforeEach  
**void** setUp(){  
 taskService = TaskService.*getInstance*();  
 id = "123456";  
 name = "add";  
 description = "testing";}

I used several software testing techniques to ensure I had adequate coverage for my programs. The first was static testing. Static testing is used to ensure that requirements and conventions are met without executing the program and to catch logical and structural errors. I used the requirements and rubric to evaluate my programs against. If any requirement was not completely met, if inconsistent standards (such as naming conventions) were found, or if a test was not accurate, the static testing was the first line of defense for cleaning up my program.

The next strategy I employed was decision table testing. Decision table testing is used to create test cases for each combination of conditions, actions, and options for business rules (What is Decision Table Testing? Explained with Example, 2023). This includes character constraints, error/success cases, and null data input rules. For each logical condition, I created a test to make sure “both sides” of each Boolean decision were tested. For example, only unique IDs could be used for each class. To test this, I created tests for duplicate IDs, ensuring an exceptions was thrown and I also created tests with unique IDs that showed no exception was thrown.

One testing technique I did not explicitly use was state transition testing. This type of testing evaluates a system as it moves from one state to another. An example of how this could be used in a business case is ‘logged in mode’ versus “logged out mode”. A logged-in customer will likely be able to see more features than a logged-out customer. Although methods for updating the contact’s name or address are state-transition adjacent operations, they were tested in other ways and didn’t really change the state of the class despite going through the transition.

When working on projects like this, where end users will eventually be using the services, it is important to be cautious of using the wrong testing tools to verify the program. As I tester, I needed to understand the purpose of each section of code so I could accurately test it (for example instead of checking if a node in the service was null, I would often need to check that it did not contain an element with a specific ID). Additionally, I found how the interdependence of different classes could affect the outcomes of tests. It was common to find a failed test for one class was actually attributed to an error in another class. For example, an error in the Task class could cause test to fail in the TaskService class.

To remove bias, I tried to allow time to pass (several days) between the programming of the classes and tests. I also tried to rely more on the decision tables I prepared for building tests than on my own opinions because they provided a more objective, logical approach to creating tests. As the developer, it can be easy to be nearsighted and build tests that address what I intended the program to do, as opposed to what it wholistically can be used to do. An example of this, which could be a future improvement, would be the lack of character validation for names and phone numbers. Validation and tests should be built to ensure that phone numbers, for example, only contain numerical characters. As a programmer, it can be easy to forget that an end user may violate this rule.

It is necessary to create high quality code and pair it with tests that do not cut any corners. According to the cost escalation model, the later an error is discovered, the more expensive it will be to resolve. If an error goes unresolved until after the software is live, it is about 1000x more expensive to fix than if it were caught during the program testing (Peter Morgan, 2019, p. 20). Additionally, there is a code of ethics software testers should consider; their work can represent management, their profession, colleagues, and the product itself (Peter Morgan, 2019, p. 37). This code of ethics may apply to certified software testers, but the implications of poorly testing programs are the same regardless of certification. I will continue to use the practices and strategies in this course to thoroughly plan tests, perform static and dynamic tests, and work in an ethical way that elevates the reputation of myself, my industry, my projects, and my colleagues.

# References

Peter Morgan, A. S. (2019). *Software Testing : An ISTQB-BCS Certified Tester Foundation guide - 4th edition.* Swindon: BCS Learning & Development Limited.

*What is Decision Table Testing? Explained with Example*. (2023, June 28). Retrieved from QAonCloud: https://www.qaoncloud.com/blog/what-is-decision-table-testing-explained-with-example