**Summary and Reflection Report**

Redacted 8/13/2025

**Summary**

My unit test design was similar in every feature that I developed in the Contact, Task and Appointment features. First, to create successful valid path tests of each constructor and service method, secondly to have more boundary and negative tests to ensure that constraints are checked. In cases of the Contact feature, I have designed tests that include [ContactTest.testValidContactCreation] to ensure that a well-formatted contact gets stored as it should, whereas unacceptable inputs like [testTooLongFirstName], [testPhoneTooLong], and [testNullAddress] checks that an argument does not follow the requirement that will cause the function to break away by throwing [illegalArgumentException]. The service layer also reflects these checks with tests such as [ContactServiceTest.testUpdateFirstName] and [testUpdateAddress], which ensures valid updates are able to continue, and tests such as [testUpdateFirstNameTooLongFails] and [testUpdateAddressNullOrTooLongFails], which verify that constraints are also being enforced on updates.

The same was done to the Task feature. Constraints were applied to this model (ID <=10, name <=20, description <=50) and both passing and failing tests were associated to it were checked, whereas the service tests both confirmed updates and prevented invalid operations. As an example, [TaskServiceTest.testUpdateTaskDescription] ensured overall updates worked, and [testUpdateNameNonexistentTaskFails] and [testAddDuplicateTaskFails] ensured that errors were handled. With the feature of Appointment, I added the time sensitivity, and therefore I would separate the date entry with the input as either future or the past and run both the test conditions. The [testValidAppointmentCreation] in [AppointmentTest] has confirmed that a valid date in the future will be accepted whereas the [testAppointmentDateInThePastThrows] has ensured that a past date will be rejected. The service also passed maximum and too-long input verification on a description length and ID length, and service tests [AppointmentServiceTest.testAddValidAppointment], [testDeleteAppointment], [testUpdateDescriptionTooLongFails] also verified proper lifecycle implementation and constraint checking.

These test fall squarely onto the written specifications of the feature: Contact rules with length of ID, name, and the phone number format and address length; Task rules including length of ID, name and description; Appointment rules that include length of ID, actual date validity and description length. Each requirement will have at least one success and one failing test so that the success and error branches are exercised. The test suite is good with an overall ratio of 86.4% project coverage: Contact package achieved an 83.7% coverage, Task package achieved an 87.2% coverage rate, and the Appointment package achieved a 90.1% coverage rate. Respectively, all the production classes (Contact, ContactService, Task, TaskService, etc.) finished with a perfect 100% coverage rate. It is also technically sound because the tests are deterministic, isolated and can be diagnosed easily in case of failure. Lastly, the suite is also fast – it utilizes in-memory data structures, minimal setup, and lacks external dependencies in result of which tests can run fast but also cover high portions of production classes.

**Reflection**

The aspect of coverage was one of the larger challenges in this projects as I also did not initially pay attention to the coverage aspect during the initial milestone assignments that partitioned this project into smaller segments. My previous mindset was to ensure the logic was working and that something had a positive and negative test, but I was not keeping an active eye on what percent of the production code was being covered. In this last build, my mindset was forced to change and I needed coverage to be a more substantial priority. This implied creating more tests to test unexplored branches but in addition, assess the validity of specific tests in terms of their usefulness. One of the barriers that I encountered was the fact that the coverage tool in Eclipse does not monitor all Lambda expressions, thus I was limited in the option of increasing coverage in those sections. I altered my priorities to fix coverage in other parts of the codebase because I did not find an efficient solution to the problem. I also deleted some of the fringe tests that were not helping much with covering the requirements and cleaned up the suite and raised the percentage on the overall coverage.

To be able to reinforce my negative test cases, I posed the question to my friends about how they would attempt to break the inputs or attempt at trying to break them should they be users. Their concepts such as the empty strings, bad phone numbers or descriptions that are too long assisted me in generating edge cases that I would not have imagined otherwise. Adding these outsider opinions meant that my tests would not just be restricted to the situations I myself thought of.

We can say the key software testing approach I exploited in this project was unit testing using JUnit 5. All tests focused on a small, focused snippet of code, like a single method or constructor, and did not depend on other sources. That helped to track down failures specifically to individual logic. The tests were deterministic, that is to say they would give the same result each time they were run and they adhered to the arrange-act-assert pattern to make them readable. I also used boundary value testing, like maximum lengths on IDs, names, descriptions, and dates within valid and invalid date ranges. As an example, the method [testInvalidDateAppointment] throws an invalid date in [AppointmentTest.testAppointmentDateInPastThrows] and the behavior of valid date is correct as determined in [testValidAppointmentCreation] in [AppointmentTest]. Negative testing was another approach utilized to make sure that the invalid inputs that were expected by the specifications generated exceptions.

Other techniques that I had not applied in this project are integration testing and mock-based testing. In this context, integration testing, which tests the effectiveness of interactions between various components, was not needed since the services in this project were autonomous and not dependent on any databases or APIs. The idea of the mock-based testing which replaces the actual dependencies with simulated ones was also not needed since data was held in the memory. In a practical scenario dealing with external systems, the techniques would be significant as these would be used to guarantee that components interact appropriately without live systems during testing.

The practical application of the techniques that I actually used is simple: unit tests with boundaries and negative samples make sure that individual methods work as expected and filter out bad data before it makes further progress into an application. This plays a crucial role in the development world since it does not allow the mistakes to continue and spread further and cost more to rectify at a later time. Techniques not adopted by me, like integration testing, are more applicable in projects where a group of modules or services relies on each other.

My thought pattern with regard to this project was conservative particularly with regard to validation logic. I ensured that each specification in the requirements was not left unchecked by having both success and failure tests. As an example, I not only tested that a valid contact phone number was accepted, but also that numbers that were too long or malformed were rejected. Such a strategy allowed me to feel the complexity of even a simple validation rule since missing only a branch might result in bugs that would not be identified.

In order to reduce bias in my appraisal of the code, I tried to treat my tests as though I was writing them for a third-party implementation. I did not presume that my own logic would be the right one; instead, I attempted to falsify it by providing edge cases and poor data. As one example, I deliberately tried to update a task that does not exist in [TaskService.testUpdateNameNonexistentTaskFails] to ensure that the service properly managed errors.

Lastly, I was disciplined in the quality of the tests. Technical debt may occur when corners are cut in the testing, and a lack of proper test coverage creates a need to fix it in an even more expensive fashion in the future. In order to prevent this, I kept coverage near 100 percent on all classes involved in production and made my tests fast enough that they could be executed frequently. This is in line with industry best practices under which high coverage, rapid execution, and meaningful test cases are used as a base upon which sustainable software quality is achieved.

**References**

Atlassian. (n.d.) *What is code coverage?* Retrieved August 13, 2025 from <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>

Arguelles, C., Ivanković, M., & Bender, A. (August 7, 2020) *Code coverage best practices*. Google Testing Blog. <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>