**Summary and Reflections Report**

1.    **Summary**

The unit test approach used to test the “Task Service,” “Contact Service,” and “Appointment Service” components of the mobile application is to first develop and write JUnit tests for the subcomponent objects (Task, Contact, and Appointment). After the subcomponents have passed tests then they are integrated into their respective components which are also individually tested.

To ensure that the test approach closely aligns with the software requirements, each component requirement has a corresponding test case in their Junit test file, and objects that require input validation also include instructions to throw an “Illegal Argument Exception” in the event of invalid input. For example, the “Contact” object in “Contact Service” is required to have 4 string attributes (ID, first name, last name, and phone) that must not be empty and must contain no more than 10 characters. The constructor method for the Contact object includes conditional statements to test each received attribute and throw an "Illegal Argument Exception" error when the input validation components are not met. The Junit test for the Contact object can then leverage the Illegal Argument Exception built into the object to test for a too-long first-name field, for example.

The conducted Junit tests are effective. Each tested requirement when ran produced the expected test result and tests that are designed to produce the wrong result worked as expected. Each of the components (“Task Service,” “Contact Service,” and “Appointment Service”) had around 67% - 82% test coverage.

As a test conducted from the perspective of a first-time test developer, the project provided a much-needed opportunity to break into the testing psychology that compliments the actual coding. Both of which is required to have full and efficient developer experience. The most effective technique used to ensure effective coding is the use of constructors with “Illegal Argument Exception” that are piped into the test codes. The testContactIdTooLong test in the ContactText.java file, for example, invokes the “Illegal Argument Exception” to test the created contact deliberated given an ID with more than 10 characters.

2.    **Reflection**

While testing the application components I utilized both black-box and white-box test techniques. Since some of the components' requirements were specified before the design and implementation of the test, the black-box techniques of equivalence partitioning and use case testing were used. A valid input equivalence partition is used to test the requirement for some fields to contain strings of specific characters. Use case testing is used in the case of requirements specifying functional requirements such as being able to generate a list of task objects, then edit and delete them. After conducting these black-box test techniques, a white-box technique was used to explore the internal structure of each component under the test to analyze test coverage.

The tests conducted so far are considered dynamic tests because they involve the use of automated tools and code implementation. Other software testing techniques that can be used are static testing and other dynamic test techniques such as boundary value analysis, decision table testing, and state transition testing.

Both static and dynamic testing are beneficial to any software development project, and some test techniques work better for some projects than others. Although the tests conducted so far are dynamic tests that help to test the production code before its released to stakeholders, static tests on deliverables of system analysis and design will also provide many benefits because they can detect errors early and offer solutions at a much lower cost.

The testing project was conducted with the mindset of an amateur tester. I was at first attempting to approach the writing test as I’ll write the code itself. An understanding of how the code-under-test and the testing code itself are both connected in the context of testing but decoupled in the context of production or deployment helped to clarify the earlier misunderstanding encountered.

It was quite difficult to limit bias in my review of my code, especially when I feel as a developer that some of the texts handled by JUnit may be carried out within the code itself. It also felt laborious and tautological to write actual code and test codes separately and an understanding that an exhaustive test is impossible all added to complicate the test biases I had. An understanding of automated testing as a tool to separate the concerns of coding and testing helped also to clarify the need for automated testing as supplied by JUnit. Writing tests into production code will make it bloated and confuse business logic with programming logic.

Finally, it is important to note that discipline and commitment to quality and well-tested codes are critical to success as a software engineering professional. Cutting corners with testing may lead to costly defects and release escapements. Although the role of testing may be delegated to another developer or team, a software developer needs to write codes with the necessary test cases aligning with the software requirements in mind. Tests must also be written to closely address software requirements and must be conducted with oversight of the tests’ coverage percentage.

**Reference:**

Brian Hambling, Peter Morgan, Angelina Samaroo, Geoff Thompson, & Peter Williams. (2019). Software Testing : An ISTQB-BCS Certified Tester Foundation Guide - 4th Edition: Vol. Fourth edition. BCS, The Chartered Institute for IT.