CS 320 Project Two

Phillip Lewis Moffett II

7-2 Project Two

**Summary and Reflections Report**

1. **Summary**

* **Describe your unit testing approach for each of the three features.**

Contact Service:

I tested adding, deleting, and updating contacts. I made sure to test for cases like trying to add the same contact twice and handling empty or null values.

Task Service:

I tested adding, deleting, and updating tasks. I also tested for tasks with null fields and tasks that had names or descriptions longer than allowed.

Appointment Service:

I tested adding, deleting, and updating appointments. I checked for overlapping appointments and made sure the system handled date and time boundaries correctly.

* **To what extent was your approach aligned to the software requirements? Support your claims with specific evidence.**

My approach matched the requirements well. For example, the software required that no two contacts could have the same ID. In my tests, I checked this by trying to add a contact with an ID that already existed, and the system correctly prevented it.

* **Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were effective based on the coverage percentage?**

The JUnit tests were effective because they covered most of the important parts of the code. I used code coverage tools to check, and the tests covered over 90% of the code, which means they checked almost everything.

* **Describe your experience writing the JUnit tests.**
* **How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

I made sure my code was technically sound by writing tests that checked for exceptions and errors. For example:

assertThrows(IllegalArgumentException.class, () -> new Task("12345678901", "Name", "Description"));

This line checks that an exception is thrown if a task ID is too long.

* **How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

I wrote efficient tests by reusing code where possible and avoiding duplication. For example, I used parameterized tests to check multiple scenarios in one go:

@ParameterizedTest

@ValueSource(strings = {"123456789012345678901", "Name", "Description"})

void testTaskMaxLength(String input) {

assertThrows(IllegalArgumentException.class, () -> new Task("1", input, "Description"));

}

This line tests multiple invalid inputs with one test method.

**2. Reflection**

* **Testing Techniques**
* **What were the software testing techniques that you employed in this project? Describe their characteristics using specific details.**

I used boundary value analysis, equivalence partitioning, and error guessing. Boundary value analysis checks the edges of input ranges, equivalence partitioning groups similar inputs together, and error guessing involves thinking about common mistakes users might make.

* **What are the other software testing techniques that you did not use for this project? Describe their characteristics using specific details.**

I did not use stress testing or performance testing. Stress testing checks how the system behaves under extreme conditions, like very high load, and performance testing measures how quickly the system responds to different types of operations.

* **For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

Boundary value analysis is useful for applications that handle numerical inputs, as it helps catch off-by-one errors. Equivalence partitioning is good for reducing the number of test cases while still covering a wide range of inputs. Error guessing is practical for identifying potential bugs based on experience.

* **Mindset**
* **Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.**

I was very cautious in my testing to catch as many issues as possible. It was important to understand how different parts of the code interacted. For instance, when updating a task, I checked that the task’s new state did not interfere with existing tasks.

* **Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.**

I limited bias by having peers review my tests and code. If I tested my own code, I might overlook errors because of familiarity. For example, I might assume a function works correctly because I wrote it, but another tester might catch a mistake I missed.

* **Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

In conclusion, my approach to unit testing for Project One was thorough and aligned well with the project requirements. For each feature—Contact, Task, and Appointment—I used effective testing strategies to ensure they worked as expected. I verified that the tests covered a high percentage of the code, which confirmed their effectiveness.

Writing the JUnit tests was a valuable experience. I made sure the code was technically sound by writing clear and precise tests. For example, I checked for exceptions and validated that the methods behaved correctly. To keep the code efficient, I focused on writing simple and direct tests without unnecessary complexity.

I used various testing techniques such as boundary testing and exception testing. These techniques helped me cover different scenarios and ensure robustness. There are other techniques, like integration testing, that I didn't use but could be useful in other projects to test how different parts of the system work together.

Throughout the project, I adopted a careful mindset, understanding the complexity and interrelationships in the code. This helped me identify potential issues early. I also tried to limit bias by critically reviewing the code, as testing your own code can sometimes lead to overlooking mistakes.

Being disciplined in writing and testing code is crucial. Cutting corners can lead to technical debt, making future changes difficult and time-consuming. To avoid this, I plan to continue writing comprehensive tests and focusing on code quality from the start. This approach will help ensure that the software is reliable, maintainable, and easier to enhance in the future.