**1. Summary**

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

For this project, my unit testing approach focused on three core features of the contact management system: creation, update, and deletion of contact objects. I used JUnit to ensure that each feature performed correctly under both normal and edge-case conditions. For the creation functionality, I validated that contacts were correctly instantiated and added to the contact service using tests like testValidContactCreation and testAddContactSuccess. To test updates, I wrote testUpdateContactSuccess and related tests to confirm that each field (first name, last name, phone number, and address) could be updated independently and that the updates followed validation rules. For deletion, I used testDeleteContactSuccess to ensure contacts could be removed and included negative tests like testDeleteContactNotFoundFails to handle invalid deletion attempts.

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

My testing approach closely followed the software requirements as described in the project guidelines. For example, in TaskTest.java, the task name field was required to be no more than 20 characters, a constraint I tested with testInvalidName. Additionally, I verified that the task description must be non-null and less than or equal to fifty characters by including tests like testUpdateTaskDescriptionInvalid. The software required that all update and delete operations should act only on existing tasks and I reflected this in testUpdateTaskDescriptionInvalid and testUpdateNonExistentTask, which test for exceptions when operations are attempted on invalid IDs.

**ii. 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?**

I ensured the quality of my JUnit tests by writing both positive and negative test cases, which allowed me to test all branches of logic in the code. Each test method had a clear purpose, and collectively, they covered the full range of input scenarios. By writing tests that validated both valid and invalid inputs, I am confident that I reached a high level of code coverage. This is because I made sure to test every constructor, method, and validation condition in all classes.

**b. Describe your experience writing the JUnit tests.**

Writing JUnit tests for this project was a valuable learning experience that reinforced my understanding of defensive programming and code validation. I focused on writing modular, clear, and isolated tests using best practices like @BeforeEach to set up reusable objects and reset the state before each test run. My goal was to create test cases that were easy to understand, yet thorough in covering all functional aspects of the contact management system.

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

To ensure my test code was technically sound, I used assertions to validate both correct outcomes and the expected exceptions. For example, I wrote assertThrows(IllegalArgumentException.class, () -> contact.setPhone("abcdefghij")) to verify that non-numeric phone numbers were properly rejected. I also used @BeforeEach to maintain consistent and isolated test environments, such as contact = new Contact("001", "Alice", "Smith", "1234567890", "123 Park Blvd");, ensuring no unintended side effects occurred between test cases.

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

Efficiency was a priority during the writing process. I avoided redundant code by using reusable setup logic in the @BeforeEach method. For example, I initialized test objects once at the start of each test, like Appointment appt = new Appointment("1", futureDate, "Dentist visit");, and then reused them in each test method as needed. I also kept each test focused on a single behavior to reduce test run time and make failures easier to diagnose.

**2. Reflection**

**a. Testing Techniques**

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

The main testing technique I used in this project was unit testing, particularly white box testing. As per GeeksforGeeks (2025), “Unit testing helps identify bugs early in the development cycle, enhance code quality, and reduce the cost of fixing issues later. It is an essential part of Test-Driven Development (TDD), promoting reliable code.” Unit testing allowed me to test individual methods and behaviors in isolation, ensuring that each component worked independently of the others. In an article, Akhtar (2024) best describes white box testing with “White Box Testing serves a crucial role in software testing by allowing testers to inspect and verify the inner workings of a software system, including its code, infrastructure, and integrations.” White box testing enabled me to write tests based on knowledge of the internal logic of the code. For instance, I specifically tested how the ContactService class handles duplicate contact IDs and invalid inputs by tracing the flow of control and exceptions.

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

Several software testing techniques were not used in this project, including integration testing, system testing, acceptance testing, and black-box testing. Integration testing focuses on interactions between modules, such as communication between the contact service and a database. System testing tests the entire application as a whole, simulating real user scenarios. Acceptance testing involves validating the system against business or user requirements, often done by stakeholders. Black-box testing evaluates software based solely on inputs and outputs, without considering the internal structure.

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

Each testing technique has a specific purpose depending on the development stage. Unit testing is most useful early in the development cycle and during test-driven development. Integration testing becomes important when multiple modules or services must interact correctly. System testing is valuable in large projects where the complete system must be validated before release. Acceptance testing ensures the final product meets stakeholder expectations and is often used in Agile methodologies. Black-box testing is well-suited for QA teams or scenarios where developers and testers are separate roles, allowing for unbiased validation.

**b. Mindset**

**i. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution?**

As a software tester, I adopted a cautious and methodical mindset. I approached each test assuming that something could go wrong and made sure to test for those failure conditions. For example, I did not assume users would always enter valid data. I tested invalid formats, null values, and overly long inputs to ensure the application could gracefully handle unexpected inputs. This cautious mindset helped me write resilient tests that anticipated real-world usage.

**ii. 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?**

Limiting bias was important throughout this project. To counteract my familiarity with the code, I intentionally wrote tests that expected failure, such as testUpdateContactNotFoundFails. These negative tests challenged assumptions I made during development and helped me uncover potential edge cases. As a developer testing my own code, I recognize that bias can be a concern because it's easy to write tests that confirm the code works, rather than tests that try to break it. Including negative tests is one way I worked to counter that bias.

**iii. 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?**

Being disciplined in maintaining code quality is critical as a software engineering professional. Cutting corners in writing or testing code may save time in the short term, but it can lead to costly bugs, system crashes, or security issues later. I plan to avoid technical debt by adhering to practices like test-driven development, continuous integration with automated tests, and regular refactoring. For example, I will always validate inputs through rigorous testing and never skip edge case handling. Committing to quality not only protects the user experience but also preserves the long-term maintainability of the codebase.

**References:**

Akhtar, H. (2024, August 12). *What is White Box Testing? (Example, Types, & Techniques) | BrowserStack*. BrowserStack. <https://www.browserstack.com/guide/white-box-testing>

GeeksforGeeks. (2025, March 26). *Unit testing software testing*. GeeksforGeeks. <https://www.geeksforgeeks.org/unit-testing-software-testing/>