**Nidhi Shree**  
**Project Two: Unit Testing Summary and Reflections**

**Summary**  
In Project One, I developed and tested the contact, task, and appointment services for a mobile application. My unit testing approach focused on verifying that each service met its defined requirements and constraints. For the contact service, I implemented tests that ensured contact IDs, names, and phone numbers followed the maximum length and non-null constraints, using both boundary tests and invalid input tests. For the task service, I created extensive tests to validate task IDs, names, and descriptions, including checks for immutability, boundary values, and invalid inputs. For the appointment service, I verified that appointment IDs, dates, and times adhered to the required formats and constraints. Each set of tests was designed to capture potential errors and confirm that the services would behave correctly in all allowed scenarios. My approach aligned closely with the software requirements because I referenced the constraints directly in my tests. For example, in TaskTest, I ensured task names did not exceed 20 characters and task IDs did not exceed 10 characters, which matches the documented requirements. Assertions such as assertThrows(Task.TaskValidationException.class, () -> new Task("", "Name", "Desc")); and assertEquals("1234567890", task.getTaskId()); were used to confirm compliance.

The overall quality of my JUnit tests is evidenced by the coverage rate, which improved from an initial 41% to above 80% after incorporating feedback. Early in the project, some edge cases and exception handling were missing, which limited coverage. After receiving feedback, I added tests for null inputs, whitespace-only strings, maximum and minimum boundary conditions, setter validation, immutability, and equality and hashcode methods. This iterative improvement ensured that nearly all logical paths and exceptions were tested. Writing the tests allowed me to develop a deep understanding of each service's behavior. I ensured technical soundness by using structured assertions and exception handling, demonstrated in lines such as assertThrows(Task.TaskValidationException.class, () -> task.setName(null)); to verify that invalid input is rejected, and assertEquals("NewName", task.getName()); to confirm proper updates. Efficiency was maintained by reusing test setup code, grouping related assertions, and avoiding redundant tests. For instance, I tested multiple boundary conditions in a single method when appropriate to reduce repetitive code while still covering all cases.

In Module Three, my Contact Service initially had only 41 percent coverage. Using the feedback provided, I improved coverage by adding robust exception handling, testing edge cases for null, empty, or oversized inputs, and verifying the add, delete, and update operations. For example, I included assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "John", "Doe", "1234567890", "123 Main St")); to check that overly long IDs were rejected. In Module Four, the Task Service coverage increased from 77% to above 80% by applying similar strategies, adding tests for duplicate tasks, boundary checks, and exception handling like assertThrows(TaskService.DuplicateTaskException.class, () -> service.addTask(task));. By the time I reached the Appointment Service, my JUnit coverage was already above 80%, but I continued to apply feedback, adding tests for invalid dates, past appointments, and excessively long descriptions. This approach ensured that each service was rigorously verified and that my tests reflected the true behavior specified in the requirements.

**Reflection**  
The primary software testing techniques I employed were unit testing, boundary testing, positive and negative testing, and exception testing. Unit testing isolated each component to ensure correct functionality, while boundary testing verified behavior at the extremes of allowable input, such as 10-character IDs or maximum description lengths. Negative testing confirmed that invalid inputs were properly rejected, and exception testing verified that informative errors were raised when constraints were violated. Other techniques not used in this project include integration testing, system testing, and performance testing. Integration testing would examine interactions between multiple services, which were not needed in this in-memory setup. System testing evaluates the full application against requirements, while performance testing measures behavior under load; both were outside the scope of this project but would be critical in production environments.

Unit testing proved essential in agile and CI/CD workflows, allowing rapid identification of defects and confidence in iterative development. My progression from 41% to 88% coverage across the three milestones demonstrates how iterative testing, combined with feedback, ensures reliable and maintainable code. Boundary testing is particularly relevant for data integrity, as in financial or healthcare applications, preventing off-by-one or truncated data errors. Negative testing is essential for user-facing applications, where invalid inputs could compromise security or functionality. Integration and system testing become vital in microservices architectures and regulatory-compliant applications, while performance testing is critical for high-traffic systems like e-commerce or streaming platforms.

The mindset I adopted emphasized caution, discipline, and objectivity. I carefully considered interdependencies between data fields, such as ensuring task IDs remained immutable when other fields were updated. Bias was mitigated by assuming my code might fail and explicitly testing expected errors, using assertions like assertThrows. Discipline was essential to avoid shortcuts, address feedback, and maintain high coverage, thereby preventing technical debt. By following this approach, I ensured that my testing not only confirmed correct behavior but also identified potential failure points before they could cause issues. My iterative improvement, guided by feedback, illustrates how careful, disciplined testing builds confidence in software quality and supports professional software engineering practices.