7-2: **Summary and Reflections Report**

### Summary

## For Project One, I created six different classes to implement the customer software requirements for a mobile scheduling application. The three primary domain classes—*Contact*, *Task*, and *Appointment*—handled the core functionality and data models. In addition to these, I developed three corresponding service classes—*ContactService*, *TaskService*, and *AppointmentService*—to manage operations such as object creation, updates, and validation logic. Together, these classes fulfilled the requirements for storing, retrieving, and managing customer contact information, task details, and appointments within the application.

My testing approach directly aligned with the software requirements provided at the beginning of the project. Each class specification clearly outlined constraints such as maximum character limits, non-null requirements, and allowed field formats. I translated each of these requirements into a set of corresponding test cases. For example, the Appointment class required a description of no more than 50 characters. I created a test in *AppointmentTest.java* to attempt object creation with a longer description, confirming that it resulted in an *IllegalArgumentException*. By mapping each requirement to one or more tests, I ensured strong alignment between testing activities and functional expectations. Throwing illegal argument exceptions was not part of the original requirements, but I added them to improve testability and help the unit tests clearly identify correct versus incorrect behavior. This proactive approach made it easier to catch errors early and ensured the code handled invalid input as expected.

I am confident in the overall quality and effectiveness of my JUnit tests because they achieved 100% code coverage across all classes. The service classes introduced additional logic for creating, updating, and deleting records, which I tested thoroughly using JUnit to verify that valid operations succeeded and invalid ones were correctly rejected. For example, in the service tests, I ensured that attempting to update a non-existent contact threw the appropriate exception, and that adding a valid task resulted in its proper storage and retrieval. The tests weren't written just to meet coverage goals—they were designed to assert meaningful behavior, validate error handling, and confirm the internal state of the services after each operation. This comprehensive testing approach ensures that both individual objects and the logic coordinating them behave as expected under a range of conditions.

I made sure my code was technically sound by writing focused, assertive test cases that clearly defined both valid and invalid outcomes. For example, in *ContactTest.java*, I used Exception ex = assertThrows(IllegalArgumentException.class, () -> {new Contact("10003", null, "Stark", "1234567890", "Stark Tower") to ensure the constructor correctly rejected null values. This confirmed that the validation logic in the constructor was doing its job and would prevent bad data from entering the system. Similarly, I used assertions like Assertions.assertEquals("Clark", contact.getFirstName()) after creating a valid object to confirm the correct state. I also ensured my tests were efficient by keeping them short and single-purpose. Rather than writing large, overloaded test cases, each test focused on one condition. For example, testPhoneTooShort() in *ContactTest.java* was strictly for checking phone numbers below the required length—nothing else. This minimized test execution time and made it easy to identify the source of any failure. In the service tests, I followed the same pattern by using clean test data and verifying results immediately after each operation, which made debugging straightforward and the code easy to maintain.

### Reflections

In this project, I employed unit testing to validate individual classes and methods in isolation. I used black-box testing to assess how classes responded to different inputs without relying on internal code, applying equivalence partitioning to separate valid inputs (like correctly formatted 10-digit phone numbers) from invalid ones (such as inputs with too few or too many digits). Boundary value analysis was also used to test edge cases—for example, verifying that a task name with exactly 20 characters was accepted while one with 21 was rejected. These techniques helped catch edge-case bugs and enforce input validation efficiently within repeatable unit tests.

Another key technique was *exception testing*, a form of negative testing, to ensure the system handled invalid data or conditions properly. Each service class included input validation to prevent issues like null fields, duplicate IDs, or updates to non-existent objects. I wrote tests to confirm these validations triggered the correct exceptions (e.g., *IllegalArgumentException)* with informative messages. Object classes had simpler unit tests focused on constructors and setters enforcing constraints such as maximum field length or valid date ranges.

While unit and exception testing fit this project’s scope, several other testing techniques were not used. *Integration testing*, which verifies interactions between modules, wasn’t needed since the project had no external systems like databases or APIs. *System testing* (end-to-end testing) was also out of scope because there was no front-end or full application workflow to test. Although I reran unit tests after changes, formal *regression testing*—usually automated in continuous integration—was not implemented. Lastly, *performance testing* wasn’t relevant for this small project but would be important if the system scaled to handle large data volumes or high traffic.

I approached this project with a cautious mindset, knowing that failures can often happen in unexpected ways. That mindset encouraged me to go beyond the obvious cases and dig deeper into testing. For example, I made a point to check for null values and overly long inputs across all fields—even when I believed the constructor would handle them. With specific requirements like a phone number needing to be exactly 10 digits and never null, I tested each condition on its own to be thorough. This level of detail helps build stronger unit tests that can hold up over time—because while I understand the code now, future changes could easily break things, and the tests need to be ready for that.

To avoid bias, I tried to think like someone else reviewing my code instead of the developer who wrote it. It’s easy to get complacent and assume everything works just because it compiles or passes a few tests. Taking that outsider perspective helped me catch edge cases I initially missed—like an empty string for a contact ID. Analyzing code coverage also gave me a clear picture of how much of the code was actually tested, helping me identify gaps and improve my test suite.

Overall, this project reminded me how important it is to stay disciplined about testing. Skipping or rushing tests might save time now but increases technical debt. To avoid technical, I focus on maintaining good coding and testing habits from the start. For example, I write unit tests alongside or even before implementing new features to catch issues early, rather than piling up bugs to fix later. I also use tools like static code analyzers to spot potential problems and enforce coding standards automatically. Incorporating continuous integration pipelines helps ensure that every code change is tested and reviewed before merging, preventing low-quality code from entering the codebase.

From my experience, rapid prototyping often prioritizes speed and flexibility over thorough testing, making it somewhat at odds with the careful, methodical approach required for effective unit testing. Rapid prototypes are designed to quickly explore ideas and validate concepts, frequently involving code that may lack comprehensive tests. This approach can accelerate development in early stages but risks introducing defects if proper testing is deferred or overlooked. Conversely, careful unit testing demands a disciplined focus on verifying individual components’ correctness, which can slow down iteration but significantly improves long-term software quality and maintainability. The best time to write unit tests is once a feature has been agreed upon or is a “core” part of the product unlikely to change. Newer features may not get tests immediately unless the schedule allows, which is something that needs clear communication with the team. Balancing these priorities requires thoughtful planning to ensure prototypes evolve into reliable, well-tested products.