**Summary and Reflection Report**

Stan Homes

Southern New Hampshire University

CS320: Software Test, Automation QA

Professor Springer

August 17, 2025

**Summary and Reflection Report**

In Project One, I developed and tested three key features for a mobile application: the ContactService, TaskService, and AppointmentService. These components formed the core of the app’s back-end functionality, enabling the creation, retrieval, updating, and deletion of essential data records. As part of Project Two, I analyzed my unit testing approach for each service, measured the effectiveness of my JUnit test through coverage results, and reflected on the techniques and mindset I applied during development. My focus was on writing clear, maintainable tests that align directly with software requirements, validate both successful and failure paths, and remain efficient for ongoing use. This report details how my testing strategies ensured the correctness, reliability, and long-term maintainability of the mobile application back-end services.

**Summary Part 1**

**Alignment to Requirements**

When I wrote my unit tests, I made sure each one tied directly to a specific requirement for the ContactService, TaskService, and AppointmentService. One of the main requirements was that each service should be able to add a new record and then retrieve it by its unique ID. I covered this in testAddAndRetrieveContact() (L13), testAddTask() (L20), and in the add portion of testAddAndDeleteAppointment() (L20). In all three cases, I added an object and then confirmed that the service returned the same instance when retrieved by ID.

Another important requirement was that each service should delete records and make sure it can’t be accessed afterward. I tested this in testDeleteContact() (L22), testDeleteTask() (L28), and in the delete portion of testAddAndDeleteAppointment() (L23). In these tests, I deleted the object and then confirmed that trying to retrieve it threw an IllegalArgumentException, which showed that the delete operation worked as intended.

The last requirement I focused on was making sure updates only changed the specific field being updated, without affecting other data. This is shown in testUpdateContact() (L31), where I only updated the first name and verified the change, and in testUpdateTaskFields() (L37–L38), where I updated both the task name and description and confirmed that both new values were saved. Connecting each requirement directly to a test like this made sure I didn’t miss any important behavior and kept a clear link between the requirements and the tests I wrote.

**Effectiveness of Tests**

I measured my test coverage using JaCoCo to make sure I was not only testing the simple parts of my code but also the parts that handle errors and special cases. The results showed 80%-line coverage. This means that most of my code, including both the main logic and the branches that handle exceptions, was executed at least once during testing. My goal was not just to reach a certain percentage but to confirm that the important features and possible failure points were all being checked.

For example, I tested adding and retrieving data, but I also checked what happens if I try to get something after it has been deleted (ContactServiceTest L22, TaskServiceTest L28, AppointmentServiceTest L23). These failure path tests are important because they prove that the services enforce their rules and do not allow invalid operations. By covering both the normal workflows and the failure scenarios, my test suite is much more likely to catch problems early and keep them from reaching production.

**Ensuring Technically Sound Code**

I made sure my tests were technically sound by keeping them consistent, focused, and tied directly to the behavior I wanted to verify. In TaskServiceTest, I used @BeforeEach (L11–L14) to create a fresh instance of the service before each test. This kept state from leaking between tests and causing false positives or negatives. I also followed a clear Arrange-Act-Assert pattern in every test. This meant setting up my data first, performing the action I wanted to test, and then checking the result with the correct assertion.

When I needed to confirm that data was stored or updated correctly, I used assertEquals to compare the expected and actual values (ContactServiceTest L13, TaskServiceTest L20, AppointmentServiceTest L20). When I needed to confirm that the service enforced rules and rejected invalid actions, I used assertThrows to check that an IllegalArgumentException was thrown (ContactServiceTest L22, TaskServiceTest L28, AppointmentServiceTest L23). These practices ensured that my tests were not just running code but proving that the code worked as intended.

**Ensuring Efficiency**

I kept my tests efficient by avoiding unnecessary setup and keeping each one focused on a single purpose. For AppointmentServiceTest, I generated a future date inline (**L**14–L18) instead of relying on a large fixture or hardcoded values. This made the test both faster to run and more reliable over time. In my update tests, I only checked the fields I intended to change, such as verifying the first name in ContactServiceTest (L30–L31) or confirming both the name and description in TaskServiceTest (L35–L38). By not adding extra, unrelated checks, I made it easier to spot exactly what failed if a test broke.

**Reflection Part 2**

**Techniques Employed**

The main techniques I used in this project were straightforward but effective for the scope of the services I was testing. I wrote happy path tests to confirm that adding, retrieving, updating, and deleting records worked exactly as expected for each service. Alongside those, I created failure path tests that deliberately tried to break the service’s rules, such as retrieving a record after it had been deleted or attempting an update with invalid data. These failure path tests are important because they prove that my services enforce their constraints instead of letting bad data through.

I also made sure to use **time-safe data generation** in my AppointmentService tests. By creating a future date at runtime instead of hardcoding one, I avoided flaky results that could fail if the test was run on a different day or year. Combined, these techniques gave me coverage of both normal workflows and the kinds of edge cases that can cause problems in real-world use.

**Techniques Not Used**

There were a few testing techniques I didn’t use for this project because they were outside the scope of unit testing or unnecessary for the current size of the codebase. I didn’t perform integration testing, since my focus here was on testing each service in isolation without worrying about how they interact together. I also didn’t do system testing or any form of end-to-end testing, which would normally be used to confirm the entire application works correctly from the user’s perspective.

Another technique I didn’t use was **property-based testing**, which can help generate a wide range of randomized inputs to uncover edge cases. This approach wasn’t needed here because the services were small and the inputs could be easily covered by targeted test cases. If these services were part of a larger system or had more complex inputs, techniques like integration testing and property-based testing would be much more valuable.

**Practical Use and Implications**

The techniques I used in this project are not only valuable for these small services but can also be applied to many different software development situations. Writing both happy path and failure path tests is useful in almost every type of application because it ensures that features work when they should and fail when they should. This is especially important in production environments where silent failures or incorrect data handling can lead to major issues.

Using time-safe data generation, like I did for the AppointmentService, is also a practical habit for any system that relies on dates or time-sensitive operations. It removes the risk of flaky tests and makes sure the results are consistent no matter when the tests are run. Similarly, keeping tests small, focused, and readable is a practice that pays off in any project because it makes the test suite easier to maintain and faster to execute as the codebase grows.

**Mindset: Caution**

While working on this project, I approached testing with the belief that even small changes could create problems if they were not caught early. I treated each service as if it could fail in unexpected ways and made sure to test both normal scenarios and error conditions. For example, in each delete test, I immediately tried to retrieve the deleted record and confirmed that it threw an exception (ContactServiceTest L22, TaskServiceTest L28, AppointmentServiceTest L23). This cautious approach helped ensure the services function correctly in both ideal and problem situations.

**Mindset: Limiting Bias**

Since I wrote the code that I was testing, I knew there was a risk of assuming it was already correct. To limit this bias, I wrote my tests as if I did not trust the implementation at all. I focused on trying to break the services by performing actions that should fail, such as retrieving a record after deletion or updating with invalid data. I also kept my assertions very direct so they would confirm whether the expected behavior happened or not. This approach helped me stay objective and ensured my tests were based on actual results rather than assumptions.

**Mindset: Discipline and Avoiding Technical Debt**

I stayed disciplined by making sure each test was purposeful, clear, and directly tied to a requirement. I avoided rushing through or skipping tests even when the code seemed straightforward. This discipline is important because cutting corners during testing often leads to bigger problems later. By keeping my tests focused and writing them alongside the features, I reduced the chances of introducing technical debt. I also plan to maintain this approach in future projects by continuing to write tests for both new features and any bug fixes. This habit will help keep the codebase stable and prevent small issues from turning into costly problems later.

**Conclusion**

This project gave me the opportunity to fully test and evaluate the ContactService, TaskService, and AppointmentService features that I built in Project One. My unit tests were closely aligned with the requirements, covered both success and failure paths, and used techniques that made them reliable and easy to maintain. I focused on keeping my code technically sound through clear structure, targeted assertions, and per-test isolation. I also maintained efficiency by keeping tests small and focused, while using practical approaches such as time-safe data generation for date-related features.

Reflecting on my work, I can see how important it is to stay cautious, limit bias, and remain disciplined when testing software. These practices not only help catch issues early but also prevent technical debt from building up over time. The methods I used here can apply to many different types of projects and will continue to be a part of my development process moving forward.