Jael Ortiz  
CS 320 – Software Test Automation QA  
16 October 2025

**Project Two: Summary and Reflections Report**

**Alignment to Requirements**

For this project, I worked on three main features: the contact service, the task service, and the appointment service. For each one, I used JUnit unit tests to check that my code met all the requirements and handled both normal and incorrect inputs.

For the contact service, my tests made sure that IDs were unique, names were not longer than 10 characters, phone numbers were exactly 10 digits, and addresses were under 30 characters. I used assertThrows to confirm that errors were raised when someone tried to create a contact with a null phone number or an ID that was too long. These tests matched the rules that were listed in the assignment.

For the task service, I tested that each task could be created, updated, and deleted the right way. I used assertEquals to make sure updates actually changed the right values. For example, when I used setName() to change the task name, I checked it with:  
assertEquals("Overtime", task3.getName());  
to confirm it updated correctly. I also checked that using invalid data, like a description over 50 characters, caused an error.

For the appointment service, I made sure that appointments couldn’t be scheduled in the past, and that every appointment had a valid ID and description. I used:  
if (appointmentDate.before(new Date()))  
to stop appointments from being created with old dates. My tests confirmed that null values or past dates threw an exception.

**Effective Tests**

All of my projects showed 80% or higher test coverage in Eclipse. For example, my contact service project showed 81.2% coverage. This means that most of my lines of code were tested, which shows that my tests were complete and covered the required cases.

**Technically Sound Code**

To keep my code technically sound, I made sure each test only focused on one part of the code at a time. I used simple names like testAddContact() or testInvalidTaskName() to make them easy to read and understand.

**Efficient Code**

I also made my tests efficient by combining checks when possible. For example:  
assertTrue(service.addContact(elmo));  
assertFalse(service.addContact(elmo));  
This tested both adding and rejecting a duplicate contact in the same test.

**Other Techniques**

The main testing techniques I used were unit testing and boundary testing.  
Unit testing means testing small parts of code, like a single method or class. This helped me find problems quickly. Boundary testing means checking values right at the limits, like 10-character names or a date exactly equal to the current day. These tests helped make sure my code didn’t break at the edges of what’s allowed.

Techniques I did not use include integration testing, system testing, and performance testing.  
Integration testing checks how different parts of a program work together, like connecting the back end to a database. System testing checks the whole program as one complete system. Performance testing checks how well the system handles heavy use. My code didn’t need those because there was no database or user interface, just small classes stored in memory.

**Uses and Implications of Techniques**

In larger software projects, these other testing methods are important. Unit and boundary testing help early in development, while integration and system testing are used later when the program is complete. Performance testing is important before releasing a product to make sure it runs smoothly under load.

**Caution**

Working on this project helped me understand what it means to think like a tester. I learned that you have to be careful and thorough, and that every small detail matters. For example, checking that an appointment date wasn’t in the past seems simple, but missing it could cause big issues for users later.

**Bias**

I also tried to avoid bias when testing my own code. It’s easy to assume that the code you wrote works perfectly, but I made myself test things I didn’t expect to fail, like using null inputs or long strings. I realized that as a developer, it’s easy to miss your own mistakes if you only test things when everything works correctly. Testing with wrong or unexpected inputs helped me catch problems I wouldn’t have seen otherwise.

**Discipline**

Being disciplined and consistent in testing is important because skipping tests or rushing through them can lead to technical debt. Fixing bugs later takes more time and money. I plan to avoid that in future projects by writing tests as I go and always checking code coverage. Keeping my code clean, readable, and well-tested will help prevent bigger issues down the line.

**Citations and Attributions**

References  
Jakubiak, N. (2025, May 2). *JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests*. Parasoft. [https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/](https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/?utm_source=chatgpt.com)