# Project Two: Summary and Reflections Report

CS-320 Software Test, Automation QA

Southern New Hampshire University

## 1. Summary

### a. Unit Testing Approach and Alignment to Requirements

For the Contact, Task, and Appointment services, I adopted a unit testing approach using JUnit 5, designed to ensure that each method within the respective service classes functioned correctly according to the software requirements. For example, in the ContactService, I tested the creation of new contact records to ensure they met field constraints like ID length, non-null names, and valid phone numbers. These requirements were derived directly from the specification document and validated through assertions such as assertNotNull, assertEquals, and assertThrows.  
  
My testing strategy aligned well with the software requirements. In the TaskService, for instance, I included tests like testAddTask\_ValidInput() and testUpdateTask\_InvalidID() to enforce validation rules specified in the original requirement set. The alignment was evident as all functional requirements—such as preventing duplicate IDs, validating string lengths, and rejecting null inputs—were directly tested.

### b. Effectiveness of JUnit Tests Based on Coverage

I measured the effectiveness of my unit tests by analyzing code coverage through my IDE’s built-in code coverage tool. Across the three services, I achieved an average coverage of approximately 95%, with the remaining 5% being primarily exception branches that were not feasible to reach due to hardcoded constraints. This level of coverage, combined with positive assertion results and catch of invalid scenarios, indicates that the test suite was both comprehensive and effective.

### c. Ensuring Technically Sound Code

To ensure the JUnit code was technically sound, I followed a structured pattern that separated test setup, action, and assertion phases. For example:  
```java  
@Test  
void testAddContact\_ValidInput() {  
 ContactService service = new ContactService();  
 Contact contact = new Contact("123", "Alice", "Smith", "1234567890", "123 Main St");  
 service.addContact(contact);  
 assertEquals("Alice", service.getContact("123").getFirstName());  
}  
```  
This test confirms that valid data is saved and retrieved as expected. I used descriptive method names, isolated test cases, and consistent naming conventions to promote clarity and maintainability.

### d. Ensuring Efficient Code

Efficiency was achieved by eliminating redundant assertions and leveraging helper methods. For instance, I created a utility method to initialize contact objects to reduce code repetition. I also avoided excessive test setup and reused objects when appropriate. Additionally, the use of @BeforeEach in JUnit allowed me to initialize shared variables efficiently across multiple test cases, reducing code duplication and improving runtime performance.

## 2. Reflection

### a. Testing Techniques Employed

During this project, I employed several fundamental testing techniques:  
- Boundary Value Testing  
- Equivalence Partitioning  
- Exception Testing  
These methods provided a well-rounded validation of both expected and edge-case behaviors in the services.

### b. Testing Techniques Not Used

I did not use mock testing or integration testing in this project. Mock testing involves simulating dependencies, which wasn't necessary here since each service was self-contained. Integration testing would typically test how services interact with one another or with external systems like databases or APIs—those were outside the scope of this unit-testing-only assignment.

### c. Practical Uses and Implications

Unit testing is ideal for test-driven development (TDD) and early detection of logic errors. Boundary and equivalence testing are useful for any input-heavy application, such as form validation. In contrast, mock and integration testing are more suited to enterprise-level systems that involve multiple interacting modules, such as e-commerce or cloud-based applications.

## 3. Mindset

### a. Caution and Complexity Awareness

I adopted a mindset of caution throughout the testing process, especially knowing that even a single untested condition could result in runtime failures. For example, when testing the AppointmentService, I ensured that IDs were tested for both uniqueness and format compliance. This caution helped me recognize the interdependence of data validation and method integrity.

### b. Limiting Bias in Code Review

To minimize bias, I reviewed each test method by asking: “If I were the user, what could go wrong?” I actively wrote tests for conditions I assumed would never happen, like passing null values or blank IDs. As a developer, it’s tempting to trust your code, but as a tester, I learned to doubt and challenge every assumption.

### c. Discipline and Avoiding Technical Debt

Being disciplined in quality assurance means resisting the urge to “just get it working.” It involves writing meaningful test cases and not skipping over negative scenarios. For instance, I intentionally added tests for removing non-existent contacts, even though such situations might seem unlikely. To avoid technical debt, I plan to incorporate testing as an ongoing habit and use automation tools like CI pipelines to ensure consistent coverage as the codebase grows.

## References

1. JUnit 5 User Guide. (n.d.). JUnit 5 Documentation. https://junit.org/junit5/docs/current/user-guide/
2. Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing (3rd ed.). Wiley.
3. Rouse, M. (2020). Technical debt. TechTarget. https://www.techtarget.com/searchsoftwarequality/definition/technical-debt