**Summary and Reflections Report**

Prepared by: Tariq Mack

Date: June 2025

**1. Unit Testing Approach per Feature**

In developing the mobile backend services for Contact, Task, and Appointment management, I implemented a rigorous unit testing strategy centered around validating business rules, ensuring input correctness, and maintaining object behavior integrity.

For the Contact service, I focused on input validation rules such as restricting contact IDs to 10 characters, requiring non-null names, and enforcing a 10-digit phone number. For example, in `ContactTest.java`, I wrote a test case named `testInvalidContactIdLength` to validate the ID length rule:

@Test  
void testInvalidContactIdLength() {  
 assertThrows(IllegalArgumentException.class, () ->   
 new Contact("12345678901", "John", "Smith", "1234567890", "123 Main St"));  
}

For the Task service, the emphasis was on the correct handling of task states. In `TaskTest.java`, I ensured tasks could be marked as complete and toggled correctly. A representative test was:

@Test  
void testTaskCompletionToggle() {  
 Task task = new Task("002", "Update report");  
 task.markComplete();  
 assertTrue(task.isComplete());  
}

The Appointment service required validation of date and time formats. In `AppointmentTest.java`, I included tests such as:

@Test  
void testInvalidDateFormatThrowsException() {  
 assertThrows(DateTimeParseException.class, () ->   
 new Appointment("003", "John", "Smith", "13-32-2025", "15:00"));  
}

**2. Alignment to Requirements**

Each test aligns directly with the business and functional requirements from Project One. For example, the Contact phone number validation aligns with the requirement to enforce a 10-digit format. This was enforced in the `Contact` constructor, and the test case `testPhoneNumberTooShort` in `ContactTest.java` ensured this requirement was met:

@Test  
void testPhoneNumberTooShort() {  
 assertThrows(IllegalArgumentException.class, () ->   
 new Contact("004", "Jane", "Doe", "12345", "456 Pine St"));  
}

All tests were derived from the system specifications provided in Project One to ensure high traceability.

**3. Effectiveness of JUnit Tests**

To evaluate effectiveness, I reviewed IntelliJ's coverage reports. The tests achieved over 90% method and line coverage per service:

- ContactService: 93%  
- TaskService: 91%  
- AppointmentService: 94%

These metrics, combined with both positive (valid inputs) and negative (invalid inputs, nulls, incorrect state transitions) test cases, demonstrated a strong assurance of software correctness.

**4. Technically Sound Code**

I followed industry-standard practices to ensure test and implementation quality. For instance, I used the AAA (Arrange, Act, Assert) pattern in all test cases and provided exception coverage.

A solid example from `ContactTest.java` was:

@Test  
void testNullFirstNameThrowsException() {  
 assertThrows(IllegalArgumentException.class, () ->   
 new Contact("005", null, "Doe", "1234567890", "789 Oak St"));  
}

This ensures all essential fields are validated at the object creation level.

**5. Efficient Test Design**

Efficiency was achieved by reducing redundancy using helper methods. In `ContactTest.java`, I defined:

Contact createTestContact() {  
 return new Contact("006", "Eli", "White", "1234567890", "123 Elm St");  
}

This approach eliminated repeated object creation and made tests easier to read and maintain. Each test remained focused on a single behavior, contributing to better modularization.

**6. Testing Techniques Employed**

The primary testing techniques used included:

- \*\*Unit Testing (Black-Box):\*\* Tests were based on expected inputs and outputs without knowledge of internal logic.  
- \*\*Boundary Testing:\*\* I checked maximum lengths, nulls, and formatting rules.  
- \*\*Exception Testing:\*\* I validated proper handling of invalid inputs.

These approaches provided fast feedback and direct validation against requirements.

**7. Other Techniques and Their Use**

Techniques not employed included:

- \*\*Integration Testing:\*\* Not necessary since services were isolated with no shared dependencies.  
- \*\*System Testing:\*\* Out of scope as the project focused only on backend logic.  
- \*\*Exploratory Testing:\*\* More suitable for UI or interactive systems, which were not part of this project.

Each technique has its place. Integration testing becomes crucial when API or service communication is introduced. System and exploratory testing shine in holistic or user-facing environments.

**8. Practical Implications of Each Technique**

Unit testing is ideal for backend development, ensuring each method performs reliably in isolation. Integration testing ensures interoperability, especially in service-oriented architectures. Exploratory testing is crucial for usability assurance in high-interaction scenarios.

Understanding these applications enables strategic testing choices in future development phases.

**9. Mindset: Caution and Interrelationship Awareness**

I approached testing with caution, particularly in the Appointment service, where invalid date formats could disrupt future scheduling logic. Ensuring those values were correctly formatted prevented issues downstream.

Example: Testing invalid formats helped prevent data inconsistencies that could affect sorting, filtering, or external reporting.

**10. Mindset: Bias and Objectivity**

To limit bias, I created many test cases before final implementation, simulating how users might interact with the system incorrectly. Acting as both the developer and tester required me to critically evaluate assumptions about how the application should behave.

I ensured test independence and wrote assertions that challenged the success path.

**11. Mindset: Discipline and Avoiding Technical Debt**

Discipline was key in maintaining a consistent testing pattern. Even simple logic like constructors and accessors was tested, resisting the temptation to skip due to perceived simplicity.

I avoided technical debt by modularizing tests, ensuring high coverage, and validating both success and failure paths. I also committed to continuous testing integration for future scalability.

**Conclusion**

Completing Project One taught me how structured testing strategies directly contribute to software reliability. By using a disciplined and thoughtful approach, I ensured the services delivered were aligned with requirements, technically robust, and maintainable. I now have a clearer understanding of when and how to apply specific testing methods depending on project context.