ASHISH RAUT

# Summary

**Unit Testing Approach for Each Feature**

In the module three, four and five milestones I implemented and tested the services handling **Contacts, Tasks** and **Appointments.** I used the JUnit framework to apply unit testing for functionality matched the required spec.

1. **Contact Service:**

There I created unit tests for successful creation, update and deletion of contacts. I tested that when contact IDs are duplicated or there are more characters than allowed in a phone number or address.

**Alignment to Requirements:**

The tests were probed directly into the business rules to avoid invalid contacts’ getting through into the system.

1. **Task Service:**

Unit tests for the task service checked the task IDs for duplicates, and normally, title and description length validation.

**Alignment to Requirements:**

I aligned test cases with module specifications by asserting failures on the invalid task inputs that were passing and confirming successes on the valid operations that were failing.

1. **Appointment Service:**

To test appointment Service had unit tests for verifying appointment ID unigue, null checks, retrieval and deletion of appointment according to the POC.

**Alignment to Requirements:**

Exactly as expected, I was able to confirm that no appointments are allowed to be scheduled with missing or duplicate IDs.

# Effectiveness of JUnit Tests

The JUnit tests were very effective.

The first milestone project exceeded 90% code coverage — most of the methods and branches were tested at least once. All kinds of assertions like assertEquals and assertThrows and assertNotNull validated both successful and failed respectively.

* Example:

assertThrows(IllegalArgumentException.class, () -> contact.setPhoneNumber("123"));

It makes sure that invalid inputs do raise correct exceptions and makes the application a little more resistant to providing it wrong data.

Quick identification of defects enabled clean, reliable code with the help of independent execution of tests after every code change.

**Experience Writing JUnit Tests**

* **Ensuring Technically Sound Code:**  
  I also made sure that objects were taken care of technically by me validating object states via assertions and exception handling.  
  Example:

assertEquals("Task Title", task.getName());

* **Ensuring Efficient Code:**  
  Since we reused(test data and fixtures) where possible, we maintained the efficiency by not having repeated setup code.  
  Example:

private final Appointment appointment = new Appointment("12345", new Date());

By doing this, tests became easier to read, maintain, and faster to execute.

**Reflection**

**Testing Techniques**

**Techniques Employed:**

* **Unit Testing with JUnit:** We tested one method at a time using JUnit software tool to confirm its individual performance.
* **Boundary Testing:** Boundary tests checked limits of character input and null limits.

**Techniques Not Used (But Important):**

* **Integration Testing:** Integration Testing was not necessary since our services operated independently but it would be vital for systems that need multiple services working together.
* **System Testing:** The complete testing of the entire system failed to take place.
* **Acceptance Testing:** Despite not being possible during development the customer check process proved its value when services went live.
* **Exploratory Testing:** The testing method I did not choose was Exploratory Testing but I know its value for usability assessment.
* **Mocking and Dependency Injection:** Mocking and Dependency Injection were not needed because external systems like databases were missing yet essential for enterprise applications.

**Practical Uses and Implications:**

* Tool for testing verifies issues before a program becomes complex.
* Integration tests confirm how actual program modules should work together.
* Customers need acceptance tests to meet their project expectations.
* Testing that goes beyond specified requirements helps detect unexpected issues that appear during product updates.
* You must mock external APIs in order to test units without environmental interference.

**Mindset**

**Caution:**

Whenever the initial tests passed, I walked the code towards services assuming there were risks hidden behind the first one. I went with the test hell of duplicate IDs, invalid lengths, null values because different minor mistakes can break the system further later.

Case in point: Database corruption could be avoided through rejecting appointments with invalid IDs.

**Bias:**

To avoid bias I took pains to write tests even before finishing my service classes where I could (similar to a Test-Driven Development approach). This also made me think in the direction of someone external questioning the code and not its creator.

**Discipline:**

I didn’t cut corners, I tested not only successful cases but deliberate failures (invalid inputs, exceptions).

From here on, this discipline is one that I will hold in moving forward with my code to avoid technical debt and to always have my code properly verified before release reducing costly rework.

# Conclusion

This project was very helpful in mastering the basics of the **unit testing** and provided insight into other necessary software quality techniques. Now that I am slowly progressing through my career, I plan to expand my testing strategy to include integration, system, acceptance and exploratory testing practices. To grow into a software engineer, I will be critical for discipline, to limit bias, and be cautious.

Building **trustworthy software** is not just testing the bugs but testing.