Unit Testing Approach

Appointment Service:

My testing approach includes various scenarios such as adding, deleting, and handling invalid appointments. Testing aimed to cover all specified requirements, including adding appointments with valid data, handling null and past dates, and ensuring unique appointment IDs.

Contact Service:

The scenarios within the Contact service include tests such as adding, deleting, and updating contacts. The tests cover all specified requirements, including adding contacts, handling duplicates, updating contact details, and handling other exceptions.

Task Service:

Each test method initializes a new instance of TaskService which scenarios such as adding, deleting, and updating tasks. The tests cover all specified requirements, including adding tasks with valid data, handling duplicates, and updating task details.

Quality of JUnit Tests

My testing approach is very effective the specified requirements in each class. As shown in my jacoco test over 80% of each class is tested, also there are various tests in my code to cover edge cases. The use of assertions was used to be sure conditions were met.

My test includes coverage of the functionality of my utility methods and getters. For example, in the code below *the testAddAppointment ensures the appointment list size is correct.* *assertEquals(1, appointmentService.getAppointments().size())*

Furthermore, my code tests for proper exception handling as shown in the code block below when a user is trying to add a null appointment.

assertThrows(IllegalArgumentException.class, () -> {

appointmentService.addAppointment(null); })

Experience Writing JUnit Tests

Technical soundness was ensured by using assertions to validate expected outcomes. For example:

assertTrue(contactService.addContact(contact)) in testAddContact ensures the contact is added.

assertFalse(contactService.addContact(contact2)) in testAddDuplicateContact ensures duplicates are not allowed.

To make my code more efficient I initialized new instances in each test method to avoid carry over effects.

TaskService taskService = new TaskService(); in each test method ensures a fresh instance.

Reflection

Testing Techniques

My class testing followed a consistent format including 3 phases of testing , unit testing, boundary testing and exception testing

Unit testing involved testing individual units of code in isolation. Ensures each method works as expected. Boundary testing included testing edge cases such as maximum and minimum input values. Exception handling ensures that methods handle invalid inputs correctly by throwing exceptions.

Given more time to make more complete testing I would integrate integration testing, system testing and acceptance testing. Integration testing to test combined parts of an application to ensure they work together. System Testing to test the complete system to verify it meets the requirements.

Mindset

I made an emphasis on employing caution by thoroughly testing edge cases and invalid inputs. An example of employing caution includes how I handle edge cases when dealing with null inputs. It was important to understand how different parts of the code interact to be sure that all parts of the code were being tested for the correct expected response. For example, ensuring that adding a duplicate appointment throws an exception.

I attempted to create unbiased testing by writing tests that cover a wide range of scenarios, not just the expected ones. For example, testing invalid phone numbers in ContactService. By remaining disciplined in my commitment to quality and avoiding shortcuts I am avoiding the concept of technical debt. Continuing to keep this commitment to quality to my code ensures a better long term stable and scalable program.