CS 320 Project Two

Laurel Horwath

04/18/25

My unit testing approach for the Contact, Task, and Appointment services focused on thorough validation of both functional behavior and input constraints. For the Contact service, I wrote unit tests to ensure the add, delete, and update operations worked based on unique IDs. I validated both success and failure paths. For example, in ContactServiceTest.java, the method testAddDuplicateContact() checks that a contact cannot be added if another contact with the same ID already exists:

assertTrue(contactService.addContact(contact1));

assertFalse(contactService.addContact(contact2));

Similarly, I created comprehensive tests in ContactTest.java for validating constructor preconditions such as null values and length violations. One example is:

Assertions.assertThrows(IllegalArgumentException.class, ()-> {

New Contact(null, “John”, “Smith”, “2104356672”, “1234 Not Real Blvd.”);

});

While I missed precondition checks in the setters originally, this feedback has helped me see how to ensure consistent data integrity.

The Junit tests I wrote were designed to provide broad code coverage. I tested positive paths (valid data) and negative paths (invalid data or operations). Although I didn’t use a specific tool to calculate code coverage, the test suite I implemented covers all the core functionality for each method. Each input condition that could raise an exception or return false is accounted for, which demonstrates a strong coverage of edge cases and business rules.

Writing the tests allowed me to better understand how the logic in each class worked and how to break that logic with improper inputs. For example, to ensure technical correctness, I used:

assertEquals(“Jane”, contactService.getContact(“1234567890”).getFirstName());

This validates that not only the method returns true but that the actual update occurred in the correct object. To ensure efficiency, I created concise, targeted tests. For example:

assertFalse(contactService.deleteContact(“1111111111”));

This directly tests the behavior without unnecessary setup.

For this project, I primarily used unit testing and equivalence partitioning. I tested normal (valid) input values and edge conditions such as nulls and strings that were too long. These helped me catch violations in constructor and update logic. Some other techniques I didn’t use include mocking and integration testing. Mocking would have helped simulate interactions between classes if they depended on external systems or services. Integration testing could validate the interaction between ContactService and the actual app UI or data persistence.In practice, mocking is useful for isolated testing of service layers, while integration testing ensures the end-to-end workflow operates as expected. These techniques are especially relevant in enterprise-scale applications or those that involve database or network calls.

Throughout the project, I approached testing with caution- making sure I validated every method in both normal and abnormal situations. For instance, I tested both null and oversized input for every field in the Contact class constructor. This mindset helped ensure robustness in my code. To avoid bias, I treated my tests as if someone else wrote the code. I didn’t assume methods would work just because I wrote them. For example, I still tested that deleting a contact actually removed it from the service- even though I was the one who implemented deleteContact(). Being disciplined in testing is essential. Cutting corners in testing can result in undetected bugs, poor user experience, and technical debt. For instance, if I hadn’t added tests for invalid phone numbers, the app could allow faulty data through. As a practitioner, I plan to write tests alongside development, ensure continuous validation through test automation, and use tools like code coverage reports to catch any gaps.