Foster Hare

CS 320

Professor Phillips

4/18/2025

Project Two

**Summary**

For the contact, task, and appointment services in this mobile application project, I used JUnit to test each service’s key features. In the contact service, I wrote tests to confirm that fields like the contact ID were within length limits. For example, I used:

assertThrows(IllegalArgumentException.class, () -> new Contact("12345678901", "Last", "1234567890", "Address"));

This made sure IDs longer than ten characters were rejected. In the task service, I tested update methods like:

taskService.updateTaskDescription("New description", taskId);

assertEquals("New description", taskService.getTask(taskId).getDescription());

These checked that task fields updated correctly. The appointment service involved handling future dates using:

Date futureDate = new GregorianCalendar(3025, Calendar.JANUARY, 1).getTime();

Appointment a = new Appointment("id", futureDate);

This allowed me to test valid appointment creation without system date issues.

My tests matched the project requirements. Each test covered a specific behavior or limit mentioned in the specs. In the contact and task services, I verified that all update methods worked and didn’t accept invalid values. In the appointment service, I confirmed that only future dates were allowed and could be updated properly.

The JUnit tests were effective because they covered the major methods and edge cases. Assertions like:

assertEquals("Sven", contactService.getContact(contactId).getFirstName());

proved that changes worked correctly. Every create, update, and delete method had a related test, giving strong coverage.

Writing the tests was smooth overall. The only challenge was dealing with date validation, which I handled using GregorianCalendar to create fixed future dates. Each test followed a clear format: setup, action, and assertion. For example:

ContactService service = new ContactService();

String contactID = service.addContact("Dr.", "Cross", "5555551111", "123 Lollypop Lane");

service.updateFirstName("Sven", contactID);

assertEquals("Sven", service.getContact(contactID).getFirstName());

This structure kept my code readable and technically sound.

**Reflection**

I mainly used unit testing, positive testing, and CRUD testing. Unit testing let me check each method in isolation. Positive testing confirmed that valid inputs worked. CRUD testing made sure each service could handle adding, viewing, editing, and deleting data.

I didn’t use negative testing, boundary testing, or equivalence partitioning. Negative testing could’ve shown how the system handled bad input. Boundary testing would’ve been helpful for fields with strict character limits. Equivalence partitioning could’ve grouped test cases to reduce repetition.

These other techniques would be useful in larger or more complex systems. Negative testing helps catch crashes or bad input handling. Boundary testing is great for edge cases. Equivalence partitioning is good for simplifying test sets when input variations grow.

**Mindset**

I approached this project carefully, assuming bugs could exist anywhere. For example, I verified that updates to fields like names and descriptions worked exactly as expected. This helped catch problems early. Understanding how the code connected also helped prevent issues, like knowing that a contact’s ID had to be correct for any updates to work.

To avoid bias while testing my own code, I reviewed the requirements often and checked my assumptions by writing extra test cases. For example, even though I trusted my constructor, I still added this test:

assertThrows(IllegalArgumentException.class, () -> new Contact(null, null, null, null));

This helped me think like someone reviewing the system, not just building it.

Being disciplined with testing is important. Skipping even small tests can lead to bugs or technical debt later. For example, not testing phone number format could cause issues if users enter the wrong length. To avoid this in the future, I’ll keep using automated testing, write tests for every new feature, and use tools like Maven to manage builds and run tests consistently.

**Conclusion**

In summary, this project showed me how essential testing is in back-end development. A careful approach, clear test cases, and focus on quality helped me build solid and dependable services that meet the customer’s needs.

**Sources**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

*JUnit - Test Framework - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. <https://www.tutorialspoint.com/junit/junit_test_framework.htm>

Jakubiak, N. (2018, August 2). JUnit Tutorial With Examples: Setting Up, Writing, and Running Java Unit Tests. Parasoft. <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>

‌