Hana Moskovitz

CS 320

Project Two

**Summary**

While developing this mobile application, I approached it by using a structured unit testing approach to make sure that the code was easily readable and that I could test the contact, task, and appointment services with the requirements that the project needed to meet. When working on Contact Service Testing, my tests checked that the customer’s contact information was stored correctly, that the program could update different fields like the name, phone number, and password, and that it handled invalid inputs safely. In the ContactServiceTest class, the testAddContactPass() method made sure that a new contact was successfully added.

When working on Task Service Testing, I focused on testing the creation, updating, and deletion of tasks. My TaskServiceTest file made sure that tasks could be added, deleted, and updated based on their ID. It also confirmed that both the name and description could be updated correctly. When working on Appointment Service Testing, my unit tests in my AppointmentServiceTest class checked that appointments were scheduled correctly and that date and time conflicts were handled as expected. The tests made sure that users could add and delete appointments and that no duplicate appointments were allowed.

**Effective Tests**

While working on coding this application, my goal was to keep my code clean and make sure that I met all of the necessary requirements. For example, with the Task Service classes, I wrote tests to make sure that the software meets the requirements. The task ID can’t be null or longer than 10 characters, so I tested that like this:

void testInvalidId() {

assertThrows(IllegalArgumentException.class, () -> {

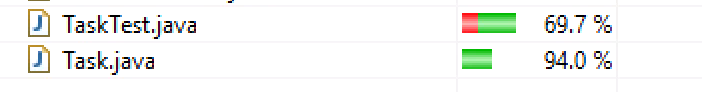
new Task(null, "Test Name", "Let's test this task.");

});

assertThrows(IllegalArgumentException.class, () -> {

new Task("123456789012", "Test Name", "Let's test this task.");

});

writing this made sure that the ID was not null or too long and that it fit with the requirements. When running the Junit tests, I was able to keep a high coverage percentage of about 90% and up. This helped me know that my code was effective. When writing my JUnit tests, I focused on making sure that my code was technically sound by testing all system requirements and checking that updates worked correctly. I used this validation in my code:

if (description == null || description.length() > 50) {

throw new IllegalArgumentException("Invalid Task Description");

}

I tested updates with this:

@Test

public void testUpdateDescription() {

Task task = new Task("123456", "Test Name", "Let's test this");

task.setDescription("A very nice test.");

assertEquals("A very nice test.", task.getDescription());

}

I also used @BeforeEach to set up my test objects so I didn’t have to repeat setup code in every test. Writing JUnit tests helped me make sure my code was correct, handled errors, and stayed clean and efficient.

**Reflection**

**Techniques Used:**

One testing technique I used throughout my milestones was unit and exception testing.

For example, in my ContactServiceTest class:

@BeforeEach

public void setUp() {

service = new ContactService();

contact = new Contact("00001", "Hana", "Moskovitz", "1234567890", "212 Muir St");

service.addContact(contact);

}

This added the contact before each test was run. I used similar @BeforeEach annotations in my AppointmentServiceTest and TaskServiceTest classes to keep things consistent and give each test a fresh start (Jakubiak, 2025). I also used Assertions.assertThrows to make sure that invalid tasks, like those with a missing or ID that are too long, correctly triggered errors.

**Techniques Not Used:**

I mostly focused on testing things case by case and stayed away from broader techniques like whole system testing or integration testing. System testing is good for checking how the entire program works together, especially before release, and integration testing checks how different parts of the program interact. I chose not to use these methods in this project because I wanted to focus directly on the project’s individual requirements and whether each part worked on its own (Pittet, n.d. ).

**Mindset**

The mindset I adopted while working on this project was one of caution and thoroughness. I wanted to make sure all requirements were met and that my tests would catch problems in my code. I tested the Task Service to make sure that invalid IDs were caught, both when the ID was null and when it was too long. I also tested that invalid updates, like setting a task name to null or making the name too long, would not be accepted after the task was created. These tests helped ensure that errors would not appear from unexpected places. It’s important to be cautious because one small issue can collide with another and cause serious problems in the system.

It was also important for me to keep my bias in check while developing and testing this project. I made sure not to assume that my code would always work and instead tested situations where it might fail. In my testUpdateNameChange method in my TaskTest class, I didn’t just test normal name updates. I also tested what would happen if someone tried to set the name to null or make it too long. This helped me catch problems I might have missed if I had only tested correct inputs. It’s important to limit bias because no one can catch everything, and unexpected errors can happen.

While developing this project, I learned the importance of being disciplined and committed to quality as a software engineer. Being fast and skipping testing might save time in the short term but can lead to serious issues and technical debt. AKA you might save time and now but it can cost you so much more in the future (Ferguson, 2024 ).If I hadn’t tested invalid inputs in my testUpdateNameChange method, I could have caused major issues with the data or even a program crash. Going forward, I plan to write good quality tests and not rush through them, even when I’m feeling pressured by deadlines. Taking the time to write solid tests now will save me from future problems.

**References**

Ferguson, A. (2024, December 19). Technical debt: The good, the bad and the manageable.

Propeller. <https://propeller.com/blog/technical-debt-the-good-the-bad-and-the-manageable>

Jakubiak, N. (2025, May 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/#How\_to\_Set\_Up\_JUnit\_Testing

Pittet , S. (n.d.). The different types of testing in software. Atlassian.

https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing