Summary and Reflections Project

For this project, I worked on building three back-end services for a mobile application: the Contact Service, Task Service, and Appointment Service. Each service required creating an object with specific requirements and constraints, along with unit tests to verify if it worked properly. I used JUnit for all the testing, focusing on writing clear and straightforward tests to make sure the code met the requirements.

My testing approach lined up directly with the software requirements the customer gave us. For example, in the Contact class, I made sure that the contact ID couldn’t be updated and stayed under 10 characters, and I wrote tests to check those exact scenarios. Same thing with the Task and Appointment classes, the tests were aimed at the specific limits the requirements mentioned, like the Task name being no more than 20 characters or making sure an Appointment date couldn’t be set in the past.

The JUnit tests I wrote were effective because they provided solid coverage of the code. I used my IDE’s built-in test coverage tool, and my coverage percentage stayed above 80% for all three services. This gave me confidence that I was testing not only normal use cases but also edge cases, like trying to create an Appointment with a null date or trying to set a Task description that was over 50 characters.

The experience writing the tests was honestly pretty smooth once I got used to the pattern. At first, I had to slow down and double-check the requirements to make sure I wasn’t missing small details. But once I had the test templates in place, it became easier to crank them out for each service. To make sure my code was technically sound, I wrote tests that covered invalid input and boundary conditions. For example, in AppointmentTest.java, I made sure to include:

java

CopyEdit

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("123", new Date(System.currentTimeMillis() - 10000), "Description");

});

This test guarantees that no appointment can be created with a date in the past, directly enforcing the requirements. I kept my code efficient by keeping the tests short and not repeating unnecessary steps. For example, in TaskServiceTest.java, I added:

java

CopyEdit

assertEquals("New Description", service.getTask("001").getDescription());

This checks the update method works in a simple line without adding clutter to the test.

For this project, the main testing technique I used was unit testing with JUnit. Unit testing focuses on individual components in isolation, which was perfect here since each service worked with its own object type (Contact, Task, or Appointment) and didn’t depend on other services. The characteristics of unit testing are that it’s fast, repeatable, and focused on specific functionality, which fit well for testing field constraints and service methods like add, delete, and update.

Other techniques like integration testing, system testing, or acceptance testing weren’t needed for this project. Integration testing looks at how different modules work together, but these services operated independently. System testing checks the whole application with all parts working, but since this project had no UI or database, there wasn’t a “full system” to test yet. Acceptance testing involves client feedback, which wasn’t present or relevant quite yet.

In bigger projects with more complex interactions or real-world scenarios, those other testing techniques are essential. For example, if the Contact Service needed to save data to a database or interact with an authentication system, integration testing would be extremely useful in avoiding errors. Similarly, acceptance testing ensures the product meets client expectations in real-world use, not just based on the code itself.

The mindset I had working on this was mostly a cautious one. I know how easy it is to overlook little details, so I paid extra attention to making sure the requirements matched my tests. It was especially important to appreciate how strict some of the requirements were, like not allowing null fields or enforcing character limits. A good example is when I wrote tests in ContactTest.java that threw exceptions for null inputs, it showed how critical it is to prevent bad data from slipping through.

Lastly, being disciplined in testing and not cutting corners is huge in this field. Sloppy code now just means I’ll have to pay for it later with bugs and issues, which just makes projects impossible to maintain. I tried to avoid that by sticking to good testing habits, writing thorough unit tests upfront, and not being lazy with edge cases. It takes a little more time, but it saves way more time (and frustration) down the line. I find that many times not being lazy actually saves you from a lot of work or aches later down the road.