**Summary**

For the Appointment feature, I wrote tests to check that appointments followed the rules for IDs, dates, and descriptions. In AppointmentTest.java, I had tests like testValidAppointment for when the contructor was initialized correctly, and testInvalidApptId for IDs that were too long. These directly tied back to the requirements that IDs can’t be over 10 characters, dates can’t be in the past, and descriptions can’t be too long.

For the Contact feature, my unit tests focused on checking names, phone numbers, and addresses. For example, in ContactTest.java, the test testContactLastNameWithMoreThanTenCharacters confirmed that the class correctly handled a name that was too long. These input length tests met the requirement that names max out at 10 characters and addresses at 30.

For the Task feature, I tested both good and bad inputs for IDs, names, and descriptions. In TaskTest.java, invalidConstructName caught the case where a name was longer than allowed, while validSetName made sure a proper name worked as expected. These input length tests met the requirement that task names can’t be more than 20 characters and descriptions can’t be more than 50.

Overall, the JUnit tests gave solid coverage. For example, in AppointmentServiceTest.java, I had both testAddAppointment for valid cases and testAddInvalidAppt for invalid ones, which meant I wasn’t just testing the valid input. By covering both sides, the tests exercised almost all the logic, giving me confidence that the main requirements were being checked thoroughly.

Writing the JUnit tests took some effort, but I made sure to keep them technically sound. One way I did this was by checking not only that an exception happened but also that the exception message was correct. In AppointmentTest.java, I used a line like assertEquals("Invalid ID", exception.getMessage()); so I knew the exact error was being thrown. That helped confirm the code was doing what it was supposed to.

I also tried to make the tests efficient by reusing setup values instead of rewriting the same things repeatedly. For instance, in TaskServiceTest.java, I reused the same name and description strings for multiple tests. In ContactServiceTest.java, I didn’t rebuild objects unnecessarily. Instead, I updated them and then used getContact(id) to verify the result. That kept the tests lean without losing coverage.

**Reflection**

The main testing technique I leaned on was equivalence partitioning. Basically, I grouped inputs into valid and invalid buckets and tested one or two examples from each group. For instance, in AppointmentTest.java, I tested a short ID like “1” (valid), a too-long ID like “01234567890” (invalid), and a null ID (invalid). That gave me solid coverage without having to test every possible value.

I also used boundary value analysis, which was especially helpful for string length limits. In ContactTest.java, I checked that a last name of exactly 10 characters worked fine, but anything longer was invalid. In TaskTest.java, I tested descriptions right at 50 characters and one past that to make sure the edge case was handled properly.

Some techniques I didn’t use were integration testing and system testing. Those would check whether different features like AppointmentService and ContactService worked together, or whether the whole app worked on a device. I also didn’t do any regression testing with automated tools. Those approaches are great for bigger projects and were beyond the scope of this assignment.

In practice, equivalence partitioning and boundary testing are great when you’re dealing with a lot of input validation, because they keep the test suite smaller but still reliable. If I were working on a large app with multiple interacting services, I’d want to add integration testing. And in a mission-critical project, I’d probably need to go even further with stress testing or formal verification.

My mindset during this project was to be careful and cover all the angles. I didn’t just test constructors, I also tested setter methods to make sure the requirements were enforced there too. For example, AppointmentTest.testInvalidNullApptDate checked both the constructor and the setter for null dates. I wanted to be sure that if one way of creating or updating the object violated the requirements, the test would catch it.

I also tried to keep myself honest and avoid bias. One way I did this was by writing some “failure” tests first, before running them against the actual code. In TaskTest.invalidConstructNullName, I expected an exception before I even confirmed the constructor logic. On the flip side, if I were only testing my own code as a developer, I might’ve been biased to just test the valid cases, so I made sure to focus on invalid cases too.

Bias is always a risk if developers only test their own work. For example, someone might assume their validation works and only write tests for valid input. By writing tests like ContactTest.testContactAddressWithMoreThanThirtyCharacters, I made sure the truncation logic was checked, even though a developer might’ve skipped that. That helped me cover areas that could’ve been overlooked.

Being disciplined about testing really mattered. If I’d cut corners, it would’ve meant missing issues like allowing past appointments or letting null names sneak into tasks. Fixing that later would’ve been messy and would add technical debt. By taking the time upfront, I reduced the chance of bugs piling up and avoided future issues.

Going forward, I plan to keep testing with both valid and invalid cases, use consistent coding practices. Just like how I checked exception messages in AppointmentTest, I’ll keep aiming for detail in my test assertions. That way I’ll avoid shortcuts, prevent technical debt, and keep codebases maintainable.