Summary & Reflection Paper

Drina Baptiste

SNHU, CS 320

Working on this project was an exciting mix of challenges and learning experiences. Let me walk you through how I approached unit testing and my overall thoughts on the process of developing the Contact, Task, and Appointment services for a mobile application.

When it came to unit testing, my focus was on making sure everything worked like a charm. For the Contact Service, I dived into validating input data—things like making sure phone numbers followed a specific pattern and names and addresses didn’t get too long. For instance, the `testAddContact` method confirmed that new contacts could be added successfully, while `testAddContactDuplicateId` ensured no funny business with duplicate IDs. I also made sure edge cases, like null or super long inputs, didn’t slip through the cracks.

The Task Service brought its own set of puzzles. I tested creating, updating, and deleting tasks while making sure input constraints were always respected. Tests like `testAddTask` and `testUpdateTaskDescription` made sure tasks behaved properly, while ensuring task names and descriptions met length limits and weren’t null. And then there was the Appointment Service, where the focus shifted to making sure dates were valid (no time travel allowed!) and descriptions weren’t empty. Tests like `testAddAppointment` and `testDeleteAppointment` covered the basics, while `testAddDuplicateAppointment` kept the ID game tight.

Throughout the process, I stayed laser-focused on the software requirements. Validation logic matched requirements to a tee, like ensuring character lengths and non-null inputs were always checked. Negative test cases—think `testDeleteNonExistentAppointment` and `testInvalidDescription`—proved that the system could gracefully handle bad inputs. And the quality of the JUnit tests? Top-notch, if I do say so myself! With near-100% coverage, I made sure both happy paths and edge cases were accounted for. The variety of assertions I used—from `assertThrows` to `assertEquals`—helped keep everything on track.

Writing the tests was a bit like solving a series of mini puzzles. One standout was the `testInvalidAppointmentDate` method, which made sure past dates got the boot. It’s a small thing, but it felt great knowing the code was robust. Efficiency-wise, I combined checks wherever it made sense. For example, in the Task Service tests, `testUpdateTaskName` validated both update and retrieval functionality in one go, keeping things clean and clear.

As for techniques, I leaned on black box testing to focus on input and output validation without getting tangled in the inner workings. Boundary value analysis helped me test edge cases like maximum string lengths, while negative testing made sure invalid inputs didn’t sneak through. Sure, I skipped white-box testing (didn’t dive into internal control flows) and performance testing (this project wasn’t about speed under load), but what I did use fit the bill perfectly.

Black-box testing works wonders for user-facing features, while white-box testing is better for heavy-duty algorithm checks. And if you’re working on real-time or high-load systems, performance testing is the MVP.

Now, let’s talk mindset. Testing code means being both a detective and a skeptic. I made sure to approach the tests assuming mistakes could happen (we’re all human, after all). For example, the duplicate ID checks in `testAddDuplicateAppointment` were there to catch any slip-ups in implementation. This mindset helped me stay objective and thorough.

Bias is always a tricky thing. To avoid it, I tried to think like a tester, not the person who wrote the code. It’s a subtle shift, but it helped me catch potential issues more effectively. On the flip side, as a developer, I can see how it’d be easy to overlook flaws in my own work. That’s why tests like `testInvalidDescription` were so satisfying—they forced me to think critically about edge cases.

In the end, staying disciplined in testing is what sets good software apart from great software. Cutting corners might save time in the short term, but it’ll cost you in the long run. Avoiding technical debt is all about being consistent, whether it’s through regular refactoring, thorough reviews, or automation.