Albert Waterman

CS 320

Project 2

Building and testing the mobile app project, my main goal was to make sure each part of the project worked exactly the way the requirements were defined. The project had three main classes, Contact, Task, and Appointment. I tested each one using JUnit to check that every rule worked and that my code responded correctly to both good and bad inputs.

For the Contact classes, I tested that the ID couldn’t be longer than 10 characters or changeable, the first and last names also stayed under 10 characters, and the phone number had to be exactly 10 digits. I also ensured the address didn’t go over 30 characters and no fields could be null. To make sure that everything worked, I created both acceptable inputs like 10-digit phone number and first name less than 10 characters (line 10 ContactTest) and bad examples like a phone number with too few digits (line 74 ContactTest) and a name that was too long (line 59 ContactTest). When testing the ContactService, I made sure duplicate IDs couldn’t be added (line 19 ContactServiceTest) , that deleting a contact removed it correctly (line 29 ContactServiceTest) , and that updates worked as expected (line 47 to 106 ContactServiceTest) .

For the Task classes, I used the same ideology. I tested that each task had a valid ID that isn’t updatable, null, and no longer than 10 characters. Name being less than 20 characters and not null. Description less than 50 characters and not null. For TaskServices I ensured you could add by TaskID, delete by TaskID, and update by TaskID with name and description updatable. As for testing, I used a very similar approach as Contact because they were efficient. I checked for null fields (Line 18-32 TaskTest), ensured setters were set (Line 62-84 TaskTest), and made sure character lengths weren’t to long (Line 40-54 TaskTest). For TaskServiceTesting I made sure that tasks could be added, updated, and deleted by Task ID. For example, I used a new task and then retrieved it using the same ID to make sure it was stored properly and also tested that trying to add another task with the same ID would cause an error, allowing me to prove that duplicate IDs weren’t allowed. (Line 12-20 TaskServiceTest). For updates, I made sure that only the task’s name and description could be changed, while the task ID remained un updatable. I also tested that deleting a task by its ID removed it from the list, and that trying to delete a task that did not exist caused an error (Line 46-54 TaskServiceTest).

The Appointment Classes focused on date and time. I made sure that the ID wasn’t updatable, null, and no longer than 10 characters. Description followed their limits of less than 50 characters and not null. Date field cannot be in the past and not null. For AppointmentService I made you able to add appointments through ID and delete appointments by ID. These tests gave me confidence that all three features worked the way they were supposed to. For the AppointmentTest, I tested that each appointment had a valid ID no longer than ten characters (Line 32 AppointmentTest), a future date (Line 24 AppointmentTest), and a description under fifty characters. I made sure null or invalid inputs, like a past date or missing description, threw the right errors. I created two small helper methods called future() and past() to make it easy to test date validation in the Appointment class. The future() method creates a date a few seconds ahead of the current time, which I used to test that valid future appointments were accepted. The past() method creates a date a few seconds before the current time, which I used to make sure the program correctly rejected past appointments (Line 14-19 AppointmentTest).

My testing approach matched the software requirements greatly because I checked every rule listed in the project’s requirements. For example, the phone number rule originally just checked for length, but I changed it to use a regular expression so that it only allowed digits. That matched the requirements exactly and made my code cleaner and more reliable. Each time I updated a rule, I also updated or added tests to prove it worked. I found this very beneficial when ensuring I got full coverage.

I know my JUnit tests were high-quality because they covered both the correct input paths and the error paths. I wrote tests that made sure valid inputs worked and that invalid ones caused the program to throw an exception. This allowed me to know I was testing every possible outcome. Since each test focused on one small thing, it was easy to find and fix problems quickly. Every time I ran my test, it covered all parts of the code constructors, methods, and even exception cases so I felt confident that I wasn’t missing anything.

Writing the JUnit tests started with simple cases to make sure the main functions worked for certain. For example, when I added a task, I wrote a test that checked if I could retrieve that same task by its ID immediately after. So, since I saw that it worked, I added tests to handle bad inputs, like trying to add two tasks with the same ID or using null for name. This allowed me to spot mistakes early and continue making progress onto the updates and deletion test methods.

I made sure my code was technically sound by testing every rule in the constructors and setters. For example, I tested phone numbers that had letters or fewer than 10 digits to make sure my validation caught them. For the services, I confirmed that adding a duplicate ID threw an error and that deleting a ID that doesn’t exist raised an exception. Each test showed that my code was catching and handling mistakes properly instead of it allowing inputs to work that didn’t make sense.

To make my code more efficient, I used a HashMap instead of an ArrayList for storing contacts, tasks, and appointments. I followed the feedback from week three and even though it was one of my first experiences with HashMap it made my programming a lot simpler. Overall, the main difference is that a HashMap can find an item instantly using its ID, while an ArrayList would have to search through every item one by one. This change made my code much faster, especially if more data were to be added. With my tests this confirmed that the data was stored and retrieved quickly and accurately. I also made sure that each test created its own new service object, so tests didn’t interfere with each other or share data.

For testing techniques, I mainly used dynamic testing, which means I ran my code to see how it behaved with different inputs. This helped me confirm that my Contact, Task, and Appointment classes worked correctly with both valid and invalid data. Dynamic testing lets me see results in real time, catch logic errors, and make sure each method follows the project requirements. I also used some static testing by reviewing my code before running it to find small issues, like typos or missing checks, which helped prevent errors early. I didn’t use integration testing or system testing because this project focused only on the back-end services, not the full app. Integration testing checks that different components work together correctly, while system testing verifies that the entire application including the front end, back end, and database runs as one complete system. Those would come later once all parts of the app are complete.

The mindset I took for this project was cautious and exciting. I made it a goal to have 100 percent coverage to ensure the safety of my code. I looked at how each class was clean and made sure one mistake wouldn’t cause a bigger issue. As an example, I double-checked that duplicate IDs couldn’t affect the update or delete functions. As far as bias, I tended to write failure tests first. This allowed me to think like a tester initially instead of a developer of my own code. This mindset helped me stay on my toes and focus on finding mistakes rather than just proving my code was correct. For example, before testing a valid phone number or task name, I first tested inputs that were too long, null, or in the wrong format. Seeing those tests fail the right way gave me confidence that my validation was working correctly.

Being disciplined about quality mattered a ton in this project. Cutting corners might save time in the short term, but it creates problems in the future. For example, if I had stuck with the slower ArrayList instead of switching to HashMap, the system could lag or crash when dealing with lots of contacts or tasks. I plan to avoid that kind of issue by always writing clean, testable code, reviewing it precisely, and running a full test after every change.