Maggie Dixon  
CS 320 - Software Testing

For this project, I used JUnit 5 to test Contact Service, Task Service, and Appointment Service. My goal was to make sure the program worked right and also followed all of the requirements. I tested different types of input: valid, invalid, and edge cases.

For the Contact Service, I made sure that each contact had a unique ID that could not be changed and that names, phone numbers, and addresses met the length limits. I also checked that the program rejected null values and incorrect formats. In the Task Service, I tested that tasks had valid IDs, names, and descriptions. I also made sure that tasks with the same ID could not be added twice and that updates only changed allowed fields. For the Appointment Service, I made sure that appointments could not be scheduled in the past and that appointment descriptions met the character limit.

When testing, I made sure that everything was in line with the requirements. I checked made sure that everything was tested for error and usage. For example, in AppointmentTest.java, I wrote a test that confirmed an error would be thrown if someone tried to create an appointment in the past:

assertThrows(IllegalArgumentException.class, () -> appt.updateDate(pastDate));

This shows that my tests aligned with the project requirements.

I also made sure that my JUnit tests were of quality by thinking of different situations they made be needed in. The tests included normal cases, incorrect inputs, and boundary values. This helped me feel more confident that my code would work. For example, in TaskServiceTest.java, I tested updating a task’s description and then checked that the update was applied:

assertTrue(testTaskService.updateDescription("2", "Updated Description"));

assertEquals("Updated Description", testTaskService.getTasks().get(1).getDescription());

This shows that my program can handle updates in the correct way.

Writing these JUnit tests was difficult, but well worth learning. At first, it was challenging to think of the different test cases, but as I worked through the project, I learned what to predict could possibly go wrong. Testing helped me find any errors before they became bigger problems and made sure my code followed the project’s requirements.

While testing my project, I used a few different testing methods. Unit testing was the main method I used, where I went and tested each class separately. This helped be able to focus on sections of the code rather than the whole thing and get confused. I also used boundary testing, where I checked values at the limit, like testing the shortest and longest allowed names. Finally, I did negative testing, where I purposefully put in incorrect values to see what would happen.

There were some testing techniques that I didn't end up using. Integration testing, which checks up on how different parts of a program work together, wasn't necessary since I tested everything separately. System testing, which tests the whole program at once, would've been useful in a bigger project, but I didn't feel like it was needed. I also did not use regression testing, which sees if old features still work after making changes, because this project did not have frequent updates. These testing methods are more common in larger projects where there's frequent updates that need to be tested often.

As a tester, I had pay attention to detail and be careful withe every move. I didn't go into it thinking everything was right, I made sure that testing was just as important as developing. This was important because even small errors can cause big problems later. For example, when testing task updates, I had to make sure only the name and description could be changed, and not the task ID. If I hadn’t tested this, I might not have noticed a mistake where IDs were accidentally being changed. This is why software engineers should always test for unexpected situations.

Avoiding technical debt is important in software development. Technical debt happens when code is written rushed without testing, which leads to errors later. Cutting corners in testing can cause bugs that are expensive to fix, and also time consuming. To prevent this, I check that my tests cover important features and that my code is structured correctly. In the future, I will continue writing strong tests, using automated tools when possible, and making sure that my code is reliable.

This project showed me how important testing is in software development. By writing JUnit tests, I was able to make sure my code met all the project requirements and worked as needed. I learned how to test for success and errors, which made my code more reliable. When I create more code, I think I will continue using unit testing to make sure my software is high quality and low on error.