Kayla Rose

CS 320

10/18/2025

Project 2

For our project I had to create Junit tests for the ContactService, TaskService, and AppointmentService classes that were built in project one to ensure that each function worked correctly. My tests focused on both expected behavior and handling invalid or unexpected inputs. For ContactService, I tested that a contact’s ID, first name, last name, phone number, and address were correctly stored and retrieved using assertions like assertEquals(“C1”, contact.getContactId()) and assertEquals(“Kayla”, contact.getFirstName()). I was able to successfully tested the getters but I did struggle to implement all the setter and some additional field tests. This did end up leaving a few untested areas and made reaching full coverage more difficult.

For the Task Class, I tested creating valid tasks with assertEquals(“1”, task.getTaskId()) and assertEquals(“TestTask”, task.getName()). I was able to verify that invalid task creation like null IDs, long IDs, null names, null descriptions, and descriptions that were exceeding the character limit, were correctly throwing IllegalArgumentException. I also tested the setter methods by changing the name and description while checking them with assertEquals to ensure that the updates were applied properly. For TaskService, I tested adding, deleting, and updating tasks. For example, I was able to confirm that a task could be added using assertTrue(service.addTask(task)) and that adding the same task again returned false. I tested deleting tasks by verifying that deleting an existing task returned true and trying to delete a non-existent task returned false. I made sure to also test updating names and descriptions while confirming that changes with assertions like assertEquals(“NewName”, service.getTask(“1”).getName()), while updates to non-existent tasks returned false. Writing these tests helped me ensure that both the Task class and the TaskService handled normal and edge-case behavior correctly.

For AppointmentService, I tested creating valid appointments with future dates using assertions like assertEquals(“12345”, appt.getAppointmentId() and assertEquals(“Checkup”, appt.getDescription()). I then tested invalid cases such as null IDs, past dates, and null descriptions, using assertThrows(IllegalArgumentException.class, () -> new Appointment(null, futureDate(), “Checkup”)). Writing these tests reinforced the importance of boundary testing for me by checking both positive and negative scenarios. Even though I was following a clear approach, achieving high coverage was sometimes challenging because some branches in constructor and exception handling were not immediately triggered by my initial tests. Using coverage reports to identify missing lines and adding targeted negative tests helped improve coverage but also gave me more confidence in the reliability of the AppointmentService.

The main testing techniques I used were unit testing and boundary testing. The unit testing allowed me to check individual methods and constructors, while boundary testing helped me to explore edge cases. For example, null inputs, duplicate IDs, and invalid dates. I did not implement integration or regression testing because this project was more focused on isolated service classes. Reflecting on this now, I think that they would have been important testing techniques that would be valuable in larger systems. Throughout our project, I maintained a cautious mindset, deliberately trying to break my own code with valid inputs rather than just confirming that it worked. This approach was useful in reducing any bias which is important when testing your own code since it can be easy to assume it is correct.

With my time spent on this project, I also learned that being disciplined in testing is critical. Skipping tests can seem faster in the short term, but this can lead to bugs, unexpected behavior, and technical debt. Using helper methods like futureDate() for appointments and @BeforeEach setup for TaskService helped reduce redundant code and made my tests cleaner. I think in the future, I plan to use test-driven development more consistently to write tests alongside code to best ensure that all new features are covered and reliable from the start.

Overall, this project showed me that testing is not just about achieving a high coverage percentage, but it is also about understanding how the code behaves in both expected and unexpected situations. Even though I struggled with coverage in ContactService and Task tests I think the process of analyzing coverage reports, identifying missing test cases, and adding negative scenarios helped me write more reliably robust code. Using concrete assertions like assertEquals, assertTrue, assertFalse, assertThrows, along with other helper methods allowed me to verify different types of behaviors across all components. This project helped strengthen my ability to write well-tested, dependable code and reinforced the importance of careful testing practices for future software development work.