Nedim Mulahusic

CS-320 Project Two

I used a structured, modular approach to unit testing for this project, focusing on the three primary features of Contacts, Appointments, and Tasks. Every feature has a specific test class to use JUnit 5 to confirm that it worked with the associated data model and service. The ultimate objective was to use thorough test coverage to guarantee data validity, method reliability, and business rule enforcement. By obtaining at least 80% coverage across all tests, I can support my claim.

I started by testing the contact feature and the contact service. To verify contactID, firstName, lastName, phone number, and address for null checks and length restrictions, ContactTest employed @Nested tests. In addition to deleting contacts and looking for unique IDs, the contactServiceTest concentrates on changing data. Since I looked for non-null fields, specified maximum character lengths, and encouraged unique IDs, the tests complied with guidelines. I validated my tests and logic using assertThrows, assertNotNull, and assedrtTrue. I then put Appointment and AppointmentService to the test. The AppointmentTest tested length validation for appointment IDs and descriptions as well as null handling using nested structures. Additionally, it confirmed that future dates are accepted; appointments cannot contain previous dates, and update methods (setDate, setDesc) appropriately handle invalid data. The AppointmentServiceTest verified that appointments could be created successfully and that duplicate IDs could be avoided. I had to follow the rules, which included future-dated appointments, non-null values, and unique IDs for appointments. To cover both valid and invalid appointment creation and make sure that exceptions are thrown, when necessary, I used JUnit tests. I tested Task and TaskService in the end. The TaskTest class used nested test groups to validate constructor validity, null handling, and length limits (ID ≤ 10, Name ≤ 20, Description < 50). The TaskServiceTest verified that object state changes are appropriately reflected through assertions and validated CRUD actions, such as adding, deleting, and updating tasks. Character restrictions, update consistency, and data authenticity were highlighted in the task-related criteria. The tests verify that the system enforces these limitations and operates correctly by simulating both valid and invalid scenarios.

Another software testing technique that I did not use was dependency checks. Dependency checks prevent runtime issues, make sure libraries and software components are working together, and identify any missing dependencies. Because dependency checks identify issues before they are executed, they would improve the speed of our programs. Integration and unit testing are two practical uses for JUnit testing. Unit testing ensures correct class and method functionality and early bug detection. Integration testing is used to test how classes interact with one another. Junit testing can be great for smaller projects to assist speed up development and for the Agile process because it can be automated. Even while dependency checks might not be required for larger projects when runtime problems can be prevented, you can still use them in smaller projects.

As a software tester, my main mindset throughout the project were analytical, thorough, and careful. I began the work with an understanding that minor problems in isolated units can lead to major system-level failures because unit testing was my primary responsibility. In addition to confirming that each app component worked as intended, I also wanted to foresee possible integration failure areas. I exercised caution by creating tests that included more than just "happy path" potential. For instance, I purposefully tested invalid scenarios like null inputs and previous dates in the AppointmentTest class. To make sure the software threw an IllegalArgumentException when constraints were violated, I also included tests for ID and name length violations in TaskTest. Instead of failing silently or causing unexpected behavior, these scenarios mimic real-world user mistakes and confirm that the system manages them correctly. Unconscious bias is a major problem in software testing, particularly when testing code that you have authored yourself. As a developer, it's normal to assume that your reasoning is sound and to unintentionally avoid testing edge scenarios that could expose errors. I approached testing with a critical, unbiased perspective to reduce bias. I purposely went over my code as if I was doing a review for someone else. For instance, just because methods like updateTask seemed simple didn't mean that they were functioning properly while I was checking the TaskService class. To verify whether the software would fail gracefully, I created tests that attempted to defy the logic, such as updating a task that didn't exist or utilizing null values. In the short term, cutting corners during testing or development may save time, but it eventually results in technical debt—hidden problems that grow and become expensive to address.

**Citations**

Mazumdar, P. (2025, May 13). *JUnit testing framework: A complete guide for beginners*. HeadSpin Blog. [https://www.headspin.io/blog/junit-a-complete-guide](https://www.headspin.io/blog/junit-a-complete-guide?utm_source=chatgpt.com) [headspin.io](https://www.headspin.io/blog/junit-a-complete-guide)