Jalen Horn

CS 320

Project 2

When I started this project, my main focus was to make sure each service, the contact, task, and appointment, worked exactly the way the requirements described. I didn’t want to just make the code run; I wanted to make sure it was reliable and followed the rules for each feature. Using JUnit tests made that possible because they allowed me to verify the code piece by piece and catch mistakes early. It helped me look at my work from a tester’s perspective instead of just a programmer’s, and that’s what improved the overall quality of the project.

Fo r the contact service, I created tests that made sure each contact had a unique ID and that all the character limits were correct. The contact ID had to be ten characters or less, the first and last names couldn’t go over ten characters, the phone number had to be exactly ten digits, and the address couldn’t exceed thirty characters. I tested both valid and invalid data to confirm that my program handled errors correctly. For example, when I used a phone number shorter than ten digits, the test failed as expected and threw an exception, which showed that my validation was accurate. These tests aligned with the project requirements because every rule listed in the assignment was covered and verified in JUnit.

For the task service, I tested that each task had a valid ID, name, and description. The name could not be longer than twenty characters, and the description had to stay within fifty. I used assertThrows to make sure that trying to add invalid or duplicate tasks caused errors. My tests also checked deleting and updating tasks to confirm that invalid operations were handled correctly. These tests proved that the service met the requirements because each condition from the rubric was validated through code.

The appointment service was a bit more advanced because it involved date validation. I verified that appointment IDs were unique, descriptions were under fifty characters, and most importantly, that the date couldn’t be in the past. I wrote tests that created both valid and invalid appointments and confirmed that the correct exceptions appeared when using past dates. That test gave me confidence that the class logic was working as intended and aligned with the rule about date validation.

Overall, the JUnit tests for all three services were effective. I reached at least 80% coverage, testing both positive and negative cases for every requirement. This proved that the project was technically sound and efficient. My approach stayed aligned with the software requirements throughout development, and every test result clearly connected back to a specific project rule. By running repeated tests and adjusting when something didn’t pass, I made sure my final code was strong and followed the expected standards.

When writing the JUnit tests, I focused on keeping my code technically sound. I used clear naming and small, separate test methods so that each one checked only one thing. This made debugging faster and made it easier to know what went wrong if a test failed. I also reused setup code when possible to make the tests efficient. I didn’t repeat logic that was already tested elsewhere, which helped keep things simple and clean. The code was efficient because it ran quickly and avoided unnecessary repetition, but it was still thorough enough to cover every scenario. By organizing my tests carefully, I could maintain clear results while reducing extra work.

The main software testing technique I used was unit testing, which focuses on testing small sections of code, such as methods or individual classes, separately. This fit perfectly for this project since I was testing each class in isolation—contact, task, and appointment—without depending on outside systems. Within my unit testing, I also used boundary value testing to check limits such as maximum character lengths and exact phone number sizes. In addition, I used exception testing to verify that invalid data triggered errors correctly. These methods were the best choice for this type of project because the classes were small, independent, and didn’t need integration with databases or user interfaces.

Some testing techniques I didn’t use included integration testing, system testing, and database testing. Integration testing focuses on how multiple systems interact, but my project didn’t include external connections. System testing checks the application as a whole, which wasn’t needed here since there was no full interface or database connection. Database testing also wasn’t used because all data was stored temporarily in memory instead of a permanent database. In larger or real-world projects, these additional techniques would be essential to verify how different systems work together, but for this specific project, unit testing was the most effective approach.

The practical uses of each testing method depend on the project’s size and structure. Unit testing is most useful during early development when testing individual logic and rules. Integration testing becomes important when different modules or APIs need to work together. System testing is key when validating the final product, especially for applications that rely on user input or databases. Understanding when to use each type of testing helps ensure quality at every stage of software development. This project helped me see that even though my focus was unit testing, each layer of testing plays an important role in professional software engineering.

During the project, I learned how important mindset and caution are when testing. I had to be disciplined enough not to assume that my code was correct just because it compiled or ran once. Instead, I treated each test as an opportunity to challenge my assumptions. I used caution when running new tests to make sure I didn’t overlook small issues that could cause bigger problems later. I also tried to avoid bias by approaching my tests as if I were testing someone else’s work. When you write the code yourself, it’s easy to think it will work perfectly, but testing forces you to see it from a different point of view. This helped me identify problems I might have ignored otherwise.

Finally, I learned that discipline is what separates average programmers from great ones. It takes effort to write thorough tests, stay organized, and not cut corners when deadlines are close. Writing quality code means being consistent, even when it’s time-consuming. This project helped me see that testing is not just an extra step, it’s part of writing the code itself. Being disciplined during this process taught me patience, structure, and accountability. I understand now that software testing is about more than finding errors; it’s about building confidence in the work you deliver.

In the end, this project gave me a much clearer understanding of how important testing is in software development. It taught me to think critically about my work, stay disciplined, and be thorough. Every successful test confirmed that I was on the right track, and every failed one taught me something valuable. That balance of patience and attention to detail is something I’ll carry forward into every coding project I do in the future.