America Sanchez-Garcia

Southern New Hampshire University

CS-320 Software Test, Automation, and QA

Dr. Kevin Tepas

October 19th, 2025

Working on this project changed the way I look at testing and debugging. I created and verified three main services, the Contact, Task, and Appointment. Each one had its own set of requirements and challenges, and the only way to handle all of them was by being consistent with my JUnit unit-testing approach. I tried to keep my testing strategy simple and focus on one thing at a time and confirmed that every part of the code matched the software requirements.

For the Contact Service, I validated five main areas. The contact ID, first name, last name, phone number, and address. Each field has strict character limits and couldn’t be null. The contact ID had to be unique and unchangeable. My JUnit tests covered every rule by using assertions like assertThrows and assertEquals. assertThrows(IllegalArgumentException.class,

() -> new Contact("12345678901", "John", "Doe", "1234567890", "123 Main St")); This verified that an ID longer than ten characters would throw an exception. I also made sure to test the boundary cases, phone numbers that were nine or eleven digits to prove the validation worked as intended.

The Task Service expanded on the structure. I tested the creation, updating, and deletion of tasks, as well as the name and description limits, assertTrue(taskService.updateTaskName("T1", "New Task")); This showed that the update method worked properly and that the system could modify one field without affecting others. Writing these tests helped me realize how small errors (like not checking for null values) can cause the entire system to behave unpredictably. On the other hand, the Appointment Service was my favorite part of this project because it introduced date validation. I had to make sure appointments couldn’t be set in the past. To handle this, I wrote a helper method using Calendar that created future dates so I didn’t have to repeat that code in every test, private Date futureDate(int days) {

Calendar c = Calendar.getInstance();

c.add(Calendar.DAY\_OF\_YEAR, days);

return c.getTime();

} This saved a lot of time and made my tests easier to read. I ran both valid and invalid cases, appointments with future dates, past dates, and null values to confirm that the logic worked across all situations.

Overall, my tests consistently hit a strong coverage level, likely around or above 90 percent. Each service’s logic, constructors, validation methods, and updates were fully tested. I ran tests repeatedly after small code changes to make sure I didn’t break anything new. Seeing all the green bars in Junit was very rewarding because it showed the application was behaving exactly how the customers’ requirements described.

Throughout this project, I mainly relied on unit testing, black-box testing, and boundary testing. Unit testing gave me control over each part of the program without worrying about outside dependencies. Black-box testing helped me focus more on the what instead of the how. I didn’t need to understand every internal logic path, I just needed to confirm the expected results. Boundary testing became really helpful for checking edge cases, such as verifying that a name or description exactly at the character limit still passed validation while anything longer failed.

There were few testing techniques I didn’t use but recognized the value of. Integration testing would have been useful if these services were connected to a database or a front-end interface, but since everything ran in memory, it wasn’t necessary yet. Regression testing would have been a smart next step if I continued adding features later, since it confirms that new code doesn’t break older functionality. Each one of these additional methods serves a different purpose and stage of development.

When I wrote my tests, I tried to think like a tester not the person who wrote the code. I assumed there were bugs, even when everything looked fine. That shift in mentality helped me become more detail oriented and realistic about how code behaves. Understanding how interconnected everything is also made me more cautious. One small mistake in a setter or validation method could ripple across an entire class. For example, if I forgot to check for null values in the Contact class, that same issue could show up later when updating a contact in the service. I also tried to stay aware of bias, It’s easy to assume your own code works just because you understand what it’s supposed to be doing. The way I avoided that was by intentionally testing scenarios like entering nulls and blank strings. When those tests failed as expected, I knew my validation logic was doing its job.