Summary and Reflections

Benjamin Cameron

Southern New Hampshire University

CS-320: Software Testing, Automation, and QA

Instructor Mathew Parker

June 29, 2025

For Project One, I created unit tests for the Contact, Task, and Appointment features using JUnit. My original approach focused on testing the basic behavior of each class, but looking back, I now realize that my approach did not fully align with the software requirements. I missed important validation checks and did not fully test edge cases, which became evident from the feedback I received and the low code coverage scores.

In the Contact class, I should have verified that the contact ID was no longer than 10 characters, that the first and last names were not null, and that the phone number was digits-only and exactly 10 characters long. I also did not test whether the address remained under 30 characters. These were all specific requirements, but I failed to include enough test cases to validate them. As a result, several critical requirements were left either unverified or untested.

The Task and Appointment classes had similar issues. I missed checks for null values, length limits, and failed to validate that appointment dates were not in the past. I also did not ensure that task descriptions stayed within the 50-character limit. Furthermore, the service classes allowed duplicate IDs, which clearly violated the project guidelines. My unit tests did not address these situations, and the feedback I received highlighted those shortcomings.

Consequently, my overall test coverage suffered. The Contact class had less than 50% coverage, and the average for all classes was around 65%, which did not meet expectations. I now realize that I should have designed my test cases to directly match the required validations outlined in the rubric. Writing more comprehensive tests could have improved both functionality and confidence in the quality of the software.

The experience of writing the JUnit tests made me more aware of how crucial it is to build a test suite that not only passes but also ensures full requirement verification. I should have used the Arrange-Act-Assert format more consistently and included both positive and negative test scenarios. Additionally, I now understand the importance of writing clear, targeted tests that check for each specific requirement.

To ensure that my code was technically sound, I should have used assertions such as 'assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "123 Main St"))' and 'assertEquals("1234567890", contact.getPhone())'. These lines would have helped me catch invalid input and verify correct values more effectively. The use of such assertions would have increased confidence in my tests and highlighted which validations were working as intended.

Reflecting on my approach, I understand that I didn’t write enough tests to cover all logic paths, particularly the ones dealing with invalid or edge-case input. Moving forward, I will be sure to write test cases that address each bullet point of the rubric and validate every required field, such as null checks, length limits, and duplicate prevention. I will also use code coverage tools earlier in the process to make sure that all necessary parts of the code are tested.

If I could go back and redo Project One, I would create tests that directly support every requirement, validate all fields properly, and organize the test cases using meaningful names and a consistent structure. I would also add more negative tests and track coverage regularly. This project made it clear that passing tests alone are not enough; you need to ensure that you are verifying exactly what the requirements demand.

This experience taught me how to be more deliberate in my testing strategies. In future projects, I will adopt a mindset that prioritizes not just functionality, but also requirement validation and test coverage. This shift in approach will help me write better code and ensure a higher level of quality and reliability in my software development work.

To build on what I’ve learned, I now recognize the importance of applying other testing techniques like integration testing and boundary value analysis. Integration testing would help ensure that features interact properly, while boundary value analysis focuses on inputs at the edges of allowed ranges. These techniques can catch additional bugs that unit tests might miss.

I also learned to approach testing with caution. For instance, using dynamic future dates instead of static ones avoids time-sensitive test failures. It’s easy to assume your code works, but being cautious means double-checking that the test actually validates what you think it does.

Bias is something I didn’t consider much before, but now I understand that writing tests only for what you expect to work leads to blind spots. To avoid bias, I made sure to include edge cases and invalid data to confirm that validation logic caught issues reliably.

Lastly, I stayed disciplined by organizing my test files, following naming conventions, and keeping my code clean and readable. Even when I felt like skipping some steps, I reminded myself that discipline now prevents issues later. This mindset will help me avoid technical debt and ensure quality in future projects.