Southern New Hampshire University

CS - 320 Project Two

Michael Ford

October 17th, 2025

**Alignment to Requirements**

For this project, I wrote JUnit tests for the Contact, Task, and Appointment services that I created earlier. My goal was to make sure every test matched the project requirements exactly and gave clear feedback about whether my code worked as intended. For example, I tested that contact IDs were always unique, that names and phone numbers met their length limits, and that appointments could not be scheduled for past dates. Each service had its own group of tests focused on adding, updating, and deleting objects. I double-checked that each test connected directly to what the assignment asked for so that the results would show, without question, if my program met the requirements. Taking the time to match each test to a requirement also helped me understand the importance of traceability, something that matters in real software projects.

**Effective Tests**

I measured how effective my tests were by checking the coverage report in Eclipse. My overall coverage was between 80% and 88%, which showed that most of my code was being tested. The TaskService had the highest coverage, around 95%, because I included extra tests for updating multiple fields and for validating edge cases. The AppointmentService was slightly lower since some parts dealt with user input that wasn’t required to be tested. Even so, I was satisfied that the main logic in all classes was covered. I also ran my tests multiple times after making small code changes to make sure nothing new broke. Seeing that my results stayed consistent gave me confidence that my tests were both accurate and dependable.

**Technically Sound Code**

When writing my JUnit tests, I focused on keeping them technically sound and easy to read. I used clear names like testAddValidContact() and deleteAppointment\_missingId\_throws() so that anyone reading the file could tell what was being tested. I relied on common assertions like assertEquals, assertTrue, and assertThrows to make sure the output matched my expectations. For example, I used assertThrows(IllegalArgumentException.class, () -> service.addTask(null)); to confirm that invalid input was rejected. Writing tests this way helped me think through each condition step by step. I also noticed that when I changed how a method handled invalid data, even small tweaks sometimes caused other tests to fail. That experience taught me how sensitive programs can be and how important it is to retest after every change.

**Efficient Code**

To make my tests efficient, I reused setup data instead of repeating the same lines of code in every test. I created helper methods to build sample contacts or tasks so I didn’t have to rewrite the same setup each time. I also grouped related tests together to keep my files organized and easy to update later. During debugging, this structure saved time because I could quickly rerun only the tests I needed. Efficiency in testing isn’t just about speed, it’s about keeping things clean and manageable as projects grow. I learned that well-organized tests make it easier to spot patterns and identify which areas might still need improvement.

**Techniques Employed**

The main testing techniques I used were unit testing, boundary testing, and equivalence partitioning. Unit testing helped me check each class individually before combining them. Boundary testing made sure data stayed within valid limits, like keeping task names under 20 characters or ensuring appointment dates were in the future. Equivalence partitioning allowed me to test just a few examples from valid and invalid ranges instead of every possible input. I liked how these techniques worked together. The unit tests handled structure, while boundary and equivalence tests focused on data accuracy. Learning to mix techniques gave me a more complete picture of how strong the code really was.

**Other Techniques**

I didn’t use integration or regression testing for this project because it was limited to individual services. Integration testing would have checked how all three services worked together, and regression testing would have helped catch new bugs after updates. If this were a real production system, I would definitely use both. For example, integration tests could confirm that a new appointment correctly links to an existing contact, while regression tests could make sure old features still work after code changes. Knowing that there are different testing layers makes me appreciate why professional teams have entire testing pipelines instead of relying only on unit tests.

**Uses and Implications of Techniques**

Each testing technique has a specific purpose. Unit testing is great for early development because it finds small bugs before they grow into bigger ones. Boundary and equivalence testing are most helpful when software takes user input, because they catch mistakes that come from unexpected values. Integration and regression tests, on the other hand, become vital in long-term projects where code keeps changing. I can see how these methods fit together in real jobs. Developers use unit tests to build confidence in their code and larger test suites to protect the system as it evolves. Understanding how and when to apply each technique will help me build stronger, more reliable software in the future.

**Caution**

While testing, I tried to be very careful with every change. Even a small edit could break something else, especially since the services shared similar validation patterns. I learned to rerun my full test suite every time I made a change, no matter how small, to catch problems early. Paying attention to how each method depended on another taught me to think like a systems engineer instead of just a programmer. One of the biggest lessons from this course was realizing that caution isn’t about fear, but to have respect for how complex code can become.

**Bias**

Testing my own code meant I had to fight against bias. It’s easy to assume your own code works just because you wrote it, but that’s when mistakes slip through. To counter that, I wrote tests designed to break my code by passing bad data, like null IDs or empty strings. I also tried to imagine what non-tech savvy users might do and test those cases. That mindset helped me see the weaknesses. In future projects, I plan to ask teammates to review my tests as well, because a second pair of eyes often finds what you miss yourself.

**Discipline**

Being disciplined with testing means finishing the process even when you think the code is fine. There were times when I wanted to move on quickly, but I made myself write the extra test to cover one more case. That extra effort paid off when I found logic errors that might have caused bugs later. I’ve learned that discipline and consistency matter more than rushing. In real-world software development, cutting corners can lead to technical debt that costs much more time to fix later. I plan to keep using version control, automated testing, and clear organization to stay disciplined and maintain quality.

**Reference**

Mausa, G., Grbac, T. G., & Basic, B. D. (2015). Data collection for software defect prediction—An exploratory case study of open-source software projects. In 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 463–469). IEEE. https://doi.org/10.1109/MIPRO.2015.7160316