Joseph Ebersole

Sherif Antoun

CS-320

20 October 2025

JUnit for Assuring Quality Code

Throughout my coursework as a computer science student, I have come to appreciate that writing code is only one part of becoming an effective software developer. The ability to test that code systematically is equally essential. In this project, I designed and implemented a suite of JUnit tests for multiple related services for the Grand Strand Systems. These services included classes like: Contact, ContactService, Task, TaskService, Appointment, and AppointmentService. My goal was not only to develop safe functionality and to verify functional correctness but also to practice professional testing habits such as precision, isolation, and disciplined execution. The AppointmentTest class, for example, tested the behavior of individual appointment objects, including their construction, validation logic, and mutator methods. The AppointmentServiceTest class extended that work by verifying how appointments were stored, retrieved, and removed within the service layer. Together, these tests represented my final iteration of unit testing for one of the services for the Grand Strand Systems. For this reflection summarizes my approach to writing the tests, the software testing techniques I used (and omitted), and my mindset while working through the process. I will also discuss why caution, attention to the relationship between classes, and commitment to disciplined testing are vital qualities for software engineers. Finally, I will reflect on how I plan to apply these lessons to avoid step backs and maintain high professional standards in the future.

In the start, My Junit testing approach began with the foundational idea that each test should be independent, repeatable, and clearly aligned with a functional requirement. I wanted my tests to verify both the happy path and possible failure conditions. This mindset guided the way I wrote each class of tests. I structured each of my test methods around three main concerns: object creation, modification, and validation. Each method used a variety of different assertions to confirm that objects behaved according to the design specifications.

While my tests were comprehensive for a small project, there were also several professional testing techniques I did not use but plan to incorporate in future work like integration testing and implementing stronger test cases. I tested each class in isolation, not as part of a larger system. Integration testing would verify that classes interact correctly, which becomes crucial in bigger applications. I also think that the core functionality coverage was too little and I could've done better to target some of the edge cases. Recognizing these omissions helped me appreciate that unit testing is just one part of a larger software quality strategy. However, focusing on unit tests gave me a strong foundation in writing scalable and fine grained tests that target the logic level.

When I first began writing JUnit tests, I underestimated the mental effort required to design good test cases. Early in the project, I would simply write tests that confirmed the code worked, but over time, I learned to write tests that could break the code intentionally. This shift from confirming correctness to challenging assumptions was one of the biggest lessons of the project. For example, when testing AppointmentService.removeAppointment(), I initially wrote tests that only removed valid entries. After running the code manually, I realized that removing a nonexistent ID threw a RuntimeException. Adding a test case for that failure condition gave me a deeper understanding of how the service handled error states. I also encountered challenges

related to test independence (thanks professor Antoun). Since AppointmentService used a singleton instance, leftover data from one test could affect another. I solved this by adding a teardown method (cleanup()) annotated with @AfterEach, which cleared the service state. This taught me the importance of test isolation which is a principle that prevents test cases from having hidden dependencies. Overall, the testing process evolved from the mechanical task into a reflective practice. Each assertion I wrote represented a design assumption, and each failure became an opportunity to improve code quality.

I feel like acting as a software tester requires a particular mindset. One that balances creativity with skepticism. I learned to think like someone who was trying to break the the code rather than an author of the code. Instead of trusting my logic, I constantly asked, “What could go wrong here?” This mindset helped me uncover subtle issues before they became defects. Caution was especially important when working with time-based data like Date objects. For example, I had to ensure that an appointment’s date could not be set in the past. Comparing timestamps (new Date(now.getTime() - 10000L)) required careful thought about how the JVM handles time and equality. Small miscalculations in milliseconds could produce misleading results.

Another area where caution mattered was the interrelationship between classes. The ContactService class depended on Contact objects behaving correctly. If the Contact class accepted invalid data, the service would unknowingly store corrupt contacts. This dependency taught me to appreciate how fragile a system can become when one layer fails validation.
By ensuring that both layers had their own validation and error handling tests, I reinforced the integrity of the system as a whole. Caution also extended to resource management. In the context of the AppointmentService class, I reset the singleton’s data after each test, and in doing so I

avoided data persistence that could cause false positives or negatives. This reinforced my understanding of test reliability; the idea that a test's outcome should depend only on the code under test, not on leftover state or prior results.

In conclusion, writing JUnit tests for the Contact, Task, then Appointment services were a challenging yet rewarding experience that solidified my understanding of software quality assurance. Through the technical requirements requested by Grand Strand Systems I learned how to design unit tests that not only confirm correctness but also challenge code assumptions through boundary and negative testing. I became familiar with essential testing techniques such as fixture management, assertion-based validation, and exception handling.

Equally important, I developed the mindset of a cautious and disciplined tester—someone who treats testing not as a chore but as an integral part of software craftsmanship. This project taught me to appreciate the interdependencies between code components, the importance of test isolation, and the value of approaching every test case with a critical eye. As I continue to grow as a software engineer, I will carry these lessons forward by committing to writing comprehensive tests, respecting the complexity of real-world systems, and maintaining professional discipline. Quality code is not achieved through luck or convenience, but it is earned through diligence and structure. This project, in many ways, was my first step toward that professional standard.

Works Cited