Project 2: Reflection

Matthew Dunfee  
Department of Computer Science: Southern New Hampshire University  
CS-320-14822-M01 Software Test, Automation QA

Omar Toledo Lopez  
April 16, 2025

My approach to unit testing was aligned with the software requirements specified in Project One. From the beginning, I knew that testing each service individually would be key to building confidence in the code and meeting expectations. For the contact service, I focused on testing validations, ensuring the contact ID and phone number followed constraints like exact length and digit-only requirements. These validations helped guarantee that only proper data could be stored. I used assertThrows with regular expressions to handle incorrect input, which allowed me to confirm that my validation logic responded properly to edge cases and violations.

For the task service, I wrote tests to separately check updating of task names and descriptions. Rather than combining update operations into one method, I followed feedback from earlier milestones and created focused update methods. Each test matched the project requirements, such as confirming updates worked as expected or verifying that errors were raised when attempting to update nonexistent task IDs. This separation of logic allowed for more targeted testing and improved clarity when debugging or expanding functionality.

When testing the appointment service, I checked date constraints to ensure appointment dates couldn't be set in the past. This was one of the trickier validations because I had to think carefully about how to check the current system time and simulate future and past values in a repeatable way. I used simple date logic within tests to meet the software requirements, applying Calendar to manipulate test date values. This allowed me to be certain that appointments reflected realistic and valid future entries.

The effectiveness of my tests was shown through the JaCoCo report, with instruction coverage of 95% and branch coverage of 84%. These numbers helped confirm the coverage and focus of my testing. Task-related tests reached 100% line coverage, showing that the methods were fully checked. This gave me confidence that the code was not only functioning as expected but that it was tested thoroughly across a variety of scenarios, including normal usage, invalid input, and edge conditions. I took time to review the report line-by-line to ensure there were no blind spots, which helped reinforce that the structure of my tests aligned with the functionality goals.

Writing JUnit tests was a step-by-step process. Early on, I had to get comfortable with the syntax and structure of the framework, but over time I developed a repeatable pattern for writing clear and consistent tests. I kept the code sound by using assertions like assertEquals to check expected results. These small checks helped prevent logical bugs from going unnoticed. To keep the code short and maintainable, I reduced repetition by using setup methods with @BeforeEach to handle common initialization across tests. That kept the actual test methods easy to read and focused on behavior rather than setup.

In this project, I used unit testing and boundary testing. Unit testing let me isolate and verify each service, which made debugging easier. Boundary testing helped find problems at input limits, like phone number length and date values. I used boundary testing especially in the Contact and Appointment classes, where character limits and date constraints required strict adherence. Having clear boundaries gave me specific targets for test input and helped build confidence that the application would handle both typical and extreme inputs gracefully.

I didn’t use integration or performance testing. Integration testing checks if components work together, which wasn't needed since modules worked separately. In the context of this assignment, each service was self-contained, and there was no persistent database or inter-process communication to validate. Performance testing, which measures response under load, wasn’t relevant for this in-memory project. However, I recognize that in a real-world system, these tests would be critical for ensuring long-term reliability and user experience.

Unit and boundary testing are useful early in development to check correctness. Integration testing is better for connected systems, and performance testing helps when apps need to handle more users or data. I’ve learned that the right testing approach depends on both the structure and the purpose of the system. Choosing tests based on the phase of development and the system's complexity is part of growing as a thoughtful software engineer.

I kept a cautious mindset during testing, thinking about how code parts could affect each other. This helped me test more scenarios, including unusual ones. For example, I made sure that updating one contact field didn’t change others. I tried to think ahead about possible ways the program could fail, especially when assumptions might be broken by user input or edge cases. Being cautious helped me avoid small mistakes that could have had larger ripple effects later.

To limit bias, I reviewed my code as if I hadn’t written it. This helped me avoid assuming the code worked. Testing each method by itself also helped reduce bias. Developers testing their own code can miss issues, which is why it’s helpful to use automated tests or involve others. That said, I know I’m not perfect. I made mistakes early on—like forcing all contact fields to be updated at once or overlooking certain validations—but after having those assignments reviewed and graded, I was able to identify the gaps and correct them in the final project. Learning from that feedback helped me take a more careful and intentional approach, especially in the final stages of testing and refinement.

Staying disciplined in how I tested and wrote code helped avoid problems. Writing full JUnit tests and aiming for good coverage showed that. These habits help reduce technical debt and make future work easier to manage. In the long run, avoiding shortcuts and building with quality in mind saves time and effort. It's a habit I plan to keep practicing as I take on larger projects and work in more complex codebases. Discipline is a foundation for growth, and quality is something I want to carry forward into everything I develop.