CS 320-18298-M01 Software Test and Automation

7-2 Journal

[CHANWOO BOK]

[[chanwoo.bok@snhu.edu](mailto:chanwoo.bok@snhu.edu)]

Southern New Hampshire University

In developing the Contact, Task, and Appointment services for the customer-facing application, I followed a structured and requirement-driven approach to unit testing using JUnit. Each service had clearly defined constraints and business logic, which I used as a baseline to write individual test cases. For example, the Contact service required that first name, last name, phone number, and address fields not be null or exceed a certain length. My tests were aligned to these requirements, ensuring that the service would reject invalid data and only allow properly formatted inputs. I created multiple JUnit test cases per requirement, targeting both normal use and edge conditions—such as extremely long strings or null values. This alignment between requirements and testing helped ensure that the services behaved reliably under various conditions.

To ensure test coverage and evaluate test effectiveness, I tracked the number of conditions and branches being tested and made sure that all logical paths were exercised. I used assertions like assertThrows to test that the services would throw exceptions when invalid inputs were passed. Additionally, I checked whether updates to objects maintained their internal integrity by using assertEquals and assertNotNull in update-related test cases. My test coverage exceeded 90%, and I believe the tests were not only broad but meaningful in validating the core functionality. The consistent and layered testing strategy gave me confidence that these services could be deployed safely and were compliant with the outlined functional specifications.

Writing the JUnit tests gave me valuable experience in structuring automated testing in a way that was both technically sound and efficient. I made use of the @BeforeEach annotation to set up test environments, ensuring that each test case had clean and independent data. This reduced repetition and improved the maintainability of my test suite. I also used helper methods to avoid redundant code, such as creating valid dummy input data to reuse across multiple test cases. This kept the test code concise and easy to extend. Technical soundness was also ensured by rigorously checking boundary conditions and exception handling, and by writing tests that mirrored real-world user input scenarios.

In terms of software testing techniques, I primarily used unit testing and white-box testing. These approaches allowed me to validate logic internal to each service without relying on external systems. I manually designed tests that covered different branches of logic, especially in input validation methods. I did not use black-box testing, integration testing, or system testing in this project, primarily because the scope was limited to individual service modules. However, I understand that these techniques would be valuable for later stages of development when the services are integrated into a full-stack application. Integration testing, for instance, would help identify communication issues between services and UI layers, while regression testing could ensure that updates to one service do not introduce bugs in others.

Working on this project, I adopted a testing mindset that emphasized caution, curiosity, and professionalism. I recognized the importance of testing as not just a task to check off, but as a quality gate for the entire application. I exercised caution by assuming that every method had the potential to fail, and designed my test cases to "break" the system. For example, I included test cases with null input, empty strings, and overly long inputs to see how the services would handle them. I also reflected on potential bias in my testing. Since I was both the developer and the tester, I was aware of the natural tendency to test only for expected behavior. To limit this bias, I created adversarial tests that challenged my own logic assumptions, helping me think more critically and prevent blind spots.

Finally, this project helped reinforce the value of discipline and quality-focused development in software engineering. It was tempting at times to stop testing once all the obvious cases passed, but I made a conscious effort to go further—to expand coverage, refactor tests, and think about long-term maintainability. As someone preparing to enter the software field, I understand that cutting corners during testing can result in technical debt, user dissatisfaction, or even security vulnerabilities. Writing clean, maintainable, and thorough tests is part of my professional identity now. This project has helped me adopt a proactive mindset where testing is not a chore but a critical component of delivering quality software.