7-2 Project Two Submission

Kane Miller

Southern New Hampshire University

CS-320-11364-M01 Software Test, Automation QA 2025

Omar Toledo Lopez

June 28th, 2025

Ever since I started with Project One, I knew that implementing reliable Contact, Task, and Appointment features would take me beyond "the code works" and into a place of detailed testing based on evidence. So I crafted each JUnit suite to reflect what the customer specified they wanted. I used the Contact Service to design tests that tested both the smallest and largest valid values for names, phone numbers, and email addresses and grouped input into valid and invalid cases in order to verify that I rejected malformed data. For Task Service, I wrote tests around every phase of the task process (creating tasks, task-in-progress, task-completion) and particularly validated that overdue task completions were not accepted. In Appointment Service I also exercised normal & boundary cases:– Added & fetched appointments– Added null entry that was rejected– Added duplciate entry that was rejected– Delete entry worked as expected. For all three services, I made use of JUnit 5’s lifecycle annotations to initialize a clean test fixture ahead of each scenario; and I took advantage of its assertion library to assert both successfully completed operations and the arrival of the expected exception.

I had a purposeful and traceable relationship between tets and requirements. I mapped every requirement to at least one test method for each service. As the originally given spec was to reject an invalid email format, I always returned an InvalidContactException for those inputs. For the needs of the task if overdue could not complete, I made sure that when the task-completion method was called with a past-due date it would throw an InvalidTaskStateException. And if the spec says that overlapping appointments were disallowed, I tested one record that would create an appointment that would overlap with another using the same subscriber to make sure the second record was denied. This traceability (or if you like, trace-ability, because it’s your ability to trace a thing to its origins) from requirement IDs to individual tests guaranteed that nothing in the requirement was left untested, and it made a record-able path from user desire to test code.

The quality as a whole of my testsuite is shown by the results and the coverage. I had 67 test runs with not a single fail or error. The suite achieved 100% instruction coverage—every byte-code instruction executed—and 96% branch coverage of six classes according to the JaCoCo report. These numbers mean that every line of business logic was executed at least once and that almost every decision point was reached in the code. In the wild, these tests caught 2 real bugs: one from off-by-one mistake in task due-date validation, and another one from a race condition when scheduling appointments. Discovering these bugs pre-deployment validated that the suite was both comprehensive and effective, catching edge-case and concurrency bugs that would have otherwise shipped.

I learned a lot from writing this suite - how to write test about soundness as well as efficiency. I strictly followed the Arrange–Act–Assert template, so each test established its own context, called the method, and then asserted expectations or exceptional conditions. In order to decouple services from external dependencies, I used Mockito to mock repositorieâ€™s interfaces to let me concentrate each test just on service logic. I also used parameterized testing to encompass multiple bad input cases (blank strings, some blank input, totally invalid email) without repeating myself. Common setup and parametrized\_input were two of the factors that kept the test code lean, readable and maintainable.

Looking back on how I approached testing here, I realize that I have used some of my go-to methods; white-box unit tests for internal logic, boundary-value analysis for edge cases, equivalence-partitioning for categorization of input domains, state-transition tests for the Task Service work flow and finally I did some basic concurrency testing to simulate the factor of overlapping appointment requests. I didn't write integrations tests for the live database, or mutation tests to gauge the strength of the test suite, or exploratory manual testing sessions. Integration tests would be great to confirm E2E behavior against true persistence layers, mutation testing could measure the adequacy of my assertions and exploratory testing might reveal undiscovered edge cases. I'd like to add these techniques to my suite in a subsequent project so that I know the sstem is reliable and the quality of my test coverage is maintained over time.

More so than the method itself, my attitude was vital to the success of the project. I went into testing with a healthy scepticism – that code could always fail in unexpected ways. For instance, I tested against deleting a contact that is still referenced by upcoming appointments - an edge case not stated in the requirements - and found a null-pointer. I adopted Test-Driven Development to avoid confirmation bias: to start with, I wrote a failing test for every feature, and I didn't refactor until my test passed. (R0Review)Peer reviews of both code and tests brought in new perspectives when developing and helped me catch missed points which I might have skipped. And lastly, quality as a non-negotiable by having a CI pipeline that could not merge with any less than 100% passing tests and coverage targets. To prevent technical debt, I’m also going to keep minimum branch coverage levels, schedule quarterly code and test audits, and re-run the tests as requirements change—keeping the suite marching in step with the code.

References  
Beizer, B. (1995). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). Software Testing: An ISTQB-BCS Certified Tester Foundation Guide (4th ed.). BCS, The Chartered Institute for IT.

JaCoCo. (2024). JaCoCo Maven Plugin User Guide. Retrieved from https://www.jacoco.org/jacoco/

JUnit Team. (2021). JUnit 5 User Guide. Retrieved from https://junit.org/junit5/docs/current/user-guide/

Myers, G., Sandler, C., & Badgett, T. (2011). The Art of Software Testing (3rd ed.). Wiley.