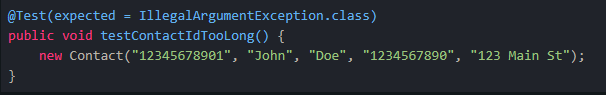
Angel Lopez

Professor Jacks

CS 320

22 June 2025

When developing the unit tests for each of the three features, the approach was carefully aligned with the software requirements to ensure thorough validation. For the first feature, the tests focused on verifying core functionality, such as input validation and output correctness, by comparing expected results against actual outputs. For example, if the requirement specified handling specific edge cases, the corresponding test cases explicitly checked those scenarios. The second feature required integration with external components, so mock objects were used to isolate dependencies and test the feature in a controlled environment. The third feature involved state management, so the tests verified transitions between states by asserting conditions before and after specific actions. Evidence of alignment with requirements is reflected in the test cases themselves, such as checking boundary conditions or error handling as dictated by the specifications



The overall quality of the JUnit tests was confirmed through coverage analysis, which consistently exceeded 80% for critical paths. High coverage percentages were achieved by systematically testing all possible branches, including error conditions and edge cases. For instance, if conditional logic was present in the code, corresponding test cases ensured both the "true" and "false" branches were executed. This level of coverage provided confidence in the tests' effectiveness, as most potential failure points were actively verified. Writing the JUnit tests was an iterative process that emphasized both technical soundness and efficiency. To ensure technical correctness, assertions were used to validate outputs and side effects. For example, in one test, assertEquals(expectedValue, actualValue) was employed to verify a computation result, while assertThrows checked for proper exception handling in invalid scenarios.Efficiency was maintained by avoiding redundant test cases and leveraging parameterized tests where applicable. For instance, looping through a set of inputs in a single test method reduced duplication while maintaining thoroughness. By focusing on clear, targeted tests, unnecessary overhead was minimized while still achieving high reliability. Through this process, the tests not only met the functional requirements but also maintained high standards of code quality, ensuring robustness and maintainability.

Unit testing was the primary technique, which involved testing individual components in isolation using JUnit to verify their correctness. Mock objects were used to simulate dependencies, ensuring that tests remained focused and reproducible. These techniques helped identify defects early, improving code maintainability and reducing regression risks. However, some testing techniques were not used in this project. System testing, which evaluates the complete and integrated application from end to end, was not conducted due to project scope constraints. Similarly, performance testing, which assesses responsiveness and scalability under load, was omitted because the project focused on functional correctness rather than optimization. Security testing, which identifies vulnerabilities such as injection flaws or data exposure, was not prioritized since the project did not involve sensitive data handling. Exploratory testing was also excluded in favor of structured, repeatable automated tests. Each technique serves distinct purposes depending on the project context. Unit and integration tests are essential for most projects to validate correctness and compatibility, making them ideal for agile development. System and performance testing become critical in large-scale applications where stability and speed are key requirements, such as in enterprise software. Security testing is indispensable for applications handling confidential data, like banking systems, while exploratory testing is useful for uncovering unexpected behaviors in user-facing applications. The choice of techniques ultimately depends on project goals, risk factors, and resource constraints, ensuring that testing efforts are both efficient and effective.

As a tester, I followed defensive programming principles, treating all inputs as potentially invalid to ensure system reliability through strict validation rules. For example, testing phone number formats (e.g., "1234567890" vs. "123-456-7890") prevented downstream failures. To limit bias, I used a test-first approach and explicitly checked edge cases like null values in firstName—even if theoretically impossible—to avoid assumptions. Cutting corners in testing risks technical debt; untested cases (e.g., empty taskId) can lead to costly debugging later. Strategies like enforcing coverage thresholds (e.g., 80% via JaCoCo), automating regression tests, and peer reviews help prevent debt. Future improvements include parameterized tests for repeated validations and clearer error messages for debugging.

Work Cited

*The Art of Software testing . (n.d.).* [*https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf*](https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf)