CS320

Aidan Farhi

2025-04-19

# CS320 Project Two

Each of the three features, contacts, tasks, and appointments, was developed with clearly defined requirements and tested using the JUnit framework. Within each respective service class, unit tests were implemented to validate that functionality aligned with constraints. For example, in the TaskServiceTest class, the method testUpdateTask() confirmed that task attributes could be updated successfully. This test specifically created a task, updated its fields, and verified the internal state change. Similarly, in the ContactServiceTest, tests validated that contact IDs were unique and not null. These test methods were tailored to reflect the requirements, such as limiting the ID to exactly ten characters and ensuring that required fields were not left empty. In the Task class, for example, the description field was required to be non-null and constrained to 1–50 characters. The corresponding test attempted to violate these constraints to verify the validation logic in the setter methods of the Task class. Each requirement had a corresponding test that proved the logic enforced constraints appropriately.

The effectiveness of the JUnit tests is also supported by high line coverage metrics. According to the IntelliJ IDEA coverage report, 100% of the code was covered by unit tests. This percentage is well above common industry thresholds and provides confidence in the correctness and completeness of the logic. There is also evidence that shows maintaining high code coverage has a positive impact on software quality. In an empirical study conducted by researchers at the Singapore Management University, it was found that there is a moderate to strong correlation between high code coverage number and the effectiveness of the test suite.[1] In testDeleteTask() (lines 32–39), the test created and deleted a task using its unique identifier and verified the deletion with assertions. Similar tests were written for all classes within each feature. Because the tests cover all code paths, including edge cases and invalid inputs, it can be asserted with confidence that the modules behave as expected.

Writing the JUnit tests was a systematic process. First, requirements were broken down into individual constraints. Then, for each constraint, test cases were written to validate both valid and invalid data. Each test method focused on a single requirement and used clear assertions to check expected outcomes. For instance, testUpdateTask() validated that after invoking updateTask(), the internal HashMap reflected the changes. By keeping tests simple and focused, issues were easier to identify and address.

Efficiency was achieved through smart use of data structures and cod reuse. Within the Task class on lines 27–46, validation was handled within setter methods, which meant that both constructors and updates benefited from centralized logic. This not only reduced redundancy but also simplified testing. Tests that invoked constructors or update methods indirectly tested the validation logic through consistent code paths. The TaskService class used a HashMap (line 7) where task identifiers served as keys and Task objects were the values. This provided constant-time complexity for key operations like get, update, and delete. This design choice reduced

[1] Shah, F., Lo, D., & Jiang, L. (2015). Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 161–170. https://ink.library.smu.edu.sg/sis_research/2974/

unnecessary overhead during test execution, and the tests confirmed the structure's performance under various conditions.

The primary testing technique used in this project was dynamic testing. Dynamic testing was applied extensively through JUnit. Unit tests were used to validate the smallest parts of each module. For example, a test ensured that a task's description could not be null and had to meet character constraints. Dynamic testing also confirmed that modules behaved correctly in edge cases, such as attempting to update or delete objects that did not exist.

Some testing techniques were not applied, such as integration testing, performance testing, and acceptance testing. There was no backend system or persistent data store, which made integration testing unnecessary. Performance testing was also excluded, since the system handled in-memory data and wasn't optimized for concurrency. Acceptance testing, which involves validating software with real users, was not applicable due to the absence of stakeholders. If used, these techniques could simulate real-world scenarios, validate system-to-system communication, and test system resilience under load.

Each testing technique has practical implications. Unit testing is best suited for early development when individual components are being built. Integration testing becomes vital when systems must communicate across interfaces. Performance testing is essential in production-facing applications, such as web services under load. Acceptance testing is critical for user-centric products, especially those in regulated industries or with customer-facing interfaces.

The mindset I adopted while working on this project was that of a cautious and detail-oriented tester. By considering each field and each operation, I was able to anticipate potential points of failure. It was important to appreciate the interdependence between methods. The validation logic inside setters was crucial both during object creation and when updating a task.

Missing this connection would have resulted in inconsistent behavior. By thinking through the lifecycle of each object, I was able to ensure robustness across scenarios.

To limit bias, I approached the tests as if they were written for someone else's code. I tried to break the logic, input invalid data, and use the services in ways that weren't initially intended. For example, I deliberately wrote test cases that attempted to add two tasks with the same ID. On the developer side, bias can arise from assuming your own code works. Writing tests that assume failure, not success, helped mitigate that.

Being disciplined in writing and testing code is paramount in professional software engineering. This crucial fact is highlighted by the IEEE Computer Society who assert that proper software testing reduces risk, increases user satisfaction, saves cost, and boosts confidence.[2] Cutting corners can lead to brittle systems, technical debt, and critical software failures. I plan to avoid this by writing modular code, including unit tests as part of every feature branch, and setting up continuous integration pipelines that enforce coverage thresholds. For example, in future projects, I plan to achieve at least 90% code coverage for all new code and automated static analysis tools as part of the build process. This proactive approach will help maintain code quality, boost confidence, and reduce future maintenance costs.

## References

Shah, F., Lo, D., & Jiang, L. (2015). Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. *Proceedings of the 2015 IEEE 22nd International*

[2] IEEE Computer Society. (n.d.). *The importance of software testing*. Retrieved April 20, 2025, from https://www.computer.org/resources/importance-of-software-testing

*Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 161–170. https://ink.library.smu.edu.sg/sis_research/2974/

IEEE Computer Society. (n.d.). *The importance of software testing*. Retrieved April 20, 2025, from https://www.computer.org/resources/importance-of-software-testing