CS 320 Project Two

Sulakshana Mehta

October 17, 2025

**Introduction**

This report provides a detailed summary and reflection on the JUnit testing process used to validate the **Contact, Task, and Appointment features** developed in Project One. The testing phase focused on ensuring correctness, reliability, and efficiency through systematic unit testing in Java using the JUnit 5 framework. Recent research emphasizes that assertions, prioritization, and automation remain essential to maintaining software quality and test reliability. This report describes the testing approach, defends the quality of the test suite, and reflects on the mindset, techniques, and discipline applied to achieve a high standard of testing practice.

**1. Summary**

**1.1 Unit Testing Approach and Alignment to Requirements**

Each core feature was tested through a dedicated JUnit test class: **ContactServiceTest**, **TaskServiceTest**, and **AppointmentServiceTest**. These tests were designed to align directly with the user stories and acceptance criteria specified in the software requirements. For example, the *addContact()* test validated input length and null restrictions to ensure compliance with functional constraints.  
According to Singh (2023), such prioritization of test cases based on requirement relevance and fault-detection potential maximizes test suite efficiency. By applying this principle, I ensured that the most critical paths, including ID validation and duplicate prevention, were verified first, improving early error detection.  
This systematic alignment reflects a best practice identified by Imran et al. (2024), who found that structured coverage analysis increases confidence in software reliability during iterative testing. By mapping each JUnit test to a functional requirement, the process ensured consistency.

**1.2 Defending Test Quality and Coverage**

The JUnit suite achieved **over 90% code coverage**, which was verified through Eclipse’s integrated coverage tool. I focused not only on numerical coverage but also on the quality of each assertion. Each test method contained multiple assertions, validating expected outputs. For instance, the *deleteContact()* method used *assertTrue()* to confirm successful deletion and *assertFalse()* to verify that the contact could no longer be retrieved. This validation follows the recommendations of Taromirad and Runeson (2025), who highlight the importance of assertion diversity in reducing false positives. Furthermore, exception handling was verified through *assertThrows(),* ensuring that the program responded correctly to invalid data. This aligns with findings by Zhang et al. (2024), who demonstrated that students who neglect rigorous assertion practices tend to produce lower-quality unit tests with limited fault detection power.

**1.3 Experience Writing JUnit Tests**

Writing the JUnit tests involved balancing clarity, independence, and reusability. To ensure technical soundness, I used the *@BeforeEach* annotation to initialize a new instance of each service before every test. Additionally, parameterized tests were used to reduce redundancy and improve maintainability. Efficiency was achieved through modular design and selective prioritization of test execution. I organized tests into logical groups and ran lightweight validation suites frequently, while running full suites during nightly builds. This approach  
provided consistent feedback without compromising project performance.

**2. Reflection**

**2.1 Testing Techniques Employed**

The primary techniques used were **black-box testing, boundary-value testing,** and **automated regression testing.** Black-box testing allowed verification of expected outputs without needing access to internal logic, ensuring an unbiased focus on behavior. Boundary-value analysis verified that system constraints (string length and date limits) were handled correctly. Automated regression testing was introduced to ensure new features did not disrupt existing functionality. Together, these methods ensured accurate and reliable testing.

**2.2 Other Techniques Not Used**

Due to time constraints, techniques such as **mutation testing and formal specification-based testing** were not implemented. Formal testing using contract-based methods like JML+JUnit could further increase precision, as explored in earlier studies (Venkatesan et al., 2020). These methods generate test cases automatically from formal specifications, reducing human error but requiring more setup. Mutation testing, while powerful for measuring test strength, was also omitted due to high computational cost. Both represent promising extensions for future testing.

**2.3 Practical Uses and Implications**

Assertion-based testing and prioritizing test cases can be useful in many types of software projects. Organizing and running tests in a planned order helps large systems scale better and allows continuous integration to run more smoothly. In addition, using clear and detailed assertions makes it easier for developers to understand and fix test failures. In real-world applications like banking or e-commerce systems, these practices help lower maintenance costs over time and make it easier to track and resolve defects.

**2.4 Mindset: Caution and Understanding of Complexity**

Testing this system required a cautious mindset, particularly due to interdependencies among features. For example, changes in the Appointment module’s date validation logic occasionally impacted Task scheduling logic, requiring retesting of both components. Understanding these interactions was essential to prevent cascading failures, since careful, reflective analysis during test design leads to stronger tests and fewer missed cases. This mindset was crucial in identifying hidden dependencies early and ensuring software reliability.

**2.5 Limiting Bias**

To minimize bias, testing responsibilities were separated from coding activities. Each test was reviewed by a peer to ensure objectivity and adherence to project requirements. Coverage reports and assertion counts were also used as quantitative metrics rather than subjective evaluations. Imran et al. (2024) show that objective metrics like code coverage and execution time provide empirical evidence of test completeness, which helped maintain fairness and objectivity throughout testing.

**2.6 Discipline and Commitment to Quality**

Maintaining discipline was critical throughout the testing phase. I adhered to consistent naming conventions, kept tests atomic, and never bypassed failing cases without investigation. According to Taromirad and Runeson (2025), disciplined assertion management fosters long-term maintainability and developer trust in test results. Likewise, Singh (2023) links consistent prioritization and organization to reduced technical debt over time. By applying these principles, I developed a habit of structured, transparent testing, and a mindset essential for sustainable software engineering practice.

**3. Conclusion**

The JUnit testing phase successfully combined systematic design, prioritization, and assertion-based validation to achieve both depth and breadth in software verification. This approach aligns closely with recent empirical evidence showing that structured coverage analysis (Imran et al., 2024), disciplined assertion use (Taromirad & Runeson, 2025), and prioritized testing (Singh, 2023) collectively improve software quality. Moreover, attention to test independence and reflection on bias mirrored best practices discussed by Zhang et al. (2024) and Venkatesan et al. (2020). Through this process, I gained a deeper understanding of testing as a technical exercise and a disciplined, evidence-based practice essential to professional software engineering.

**References**

Imran, M., Cortellessa, V., Di Ruscio, D., Rubei, R., & Traini, L. (2024). *An empirical study on code coverage of performance testing.* In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE ’24)* (pp. 48–57). Association for Computing Machinery. <https://doi.org/10.1145/3661167.3661196>

Singh, A. (2023). *A systematic literature review on test case prioritization.* In *Handbook on Test Case Prioritization* (Chap. 7). Wiley. [https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119896838.ch7](https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119896838.ch7?utm_source=chatgpt.com)

Taromirad, M., & Runeson, P. (2025). *Assertions in software testing: Survey, landscape, and trends.* *International Journal on Software Tools for Technology Transfer, 27,* 117–135. [https://link.springer.com/article/10.1007/s10009-025-00794-1](https://link.springer.com/article/10.1007/s10009-025-00794-1?utm_source=chatgpt.com)

Venkatesan, P. K., Rozario, R. G., & Fiaidhi, J. (2020, April 13). *JUnit framework for unit testing.* *TechRxiv.* <https://doi.org/10.36227/techrxiv.12092259.v1>

Zhang, S., Yang, Y., & Wang, Z. (2024). *Why students write low-quality unit testing scripts: An empirical study from software testing skill competitions.* In *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)* (pp. 719–720). IEEE. <https://doi.org/10.1109/QRS-C63300.2024.00095>