**Project Two Submission**

Zachary N. Locke

Southern New Hampshire University

CS-320 Software Test, Automation

Jeff Phillips

April 20, 2025

This report provides a detailed summary and reflection on the unit testing strategies and outcomes for the Contact, Task, and Appointment management application developed during Project One. This report focuses on how unit testing techniques were applied to meet software requirements, evaluate the effectiveness and quality of the JUnit test implementations, and reflect on the methods and mindset adopted throughout the software testing process. This summary is structured around the competencies of technical accuracy, alignment with requirements, disciplined testing behavior, and bias mitigation in software quality assurance. Each test case was directly derived from functional requirements laid out in the rubric. For example, the Contact class required strict validation such as a contact ID of no more than 10 characters, a phone number of exactly 10 digits, and fields like firstName, lastName, and address that must not be null or empty. These constraints were explicitly verified in ContactTest.java through assertions like assertThrows(IllegalArgumentException.class, ...) for invalid inputs and assertEquals(...) for valid cases. Similar coverage was applied to the Task and Appointment classes and their corresponding services. This ensured that test cases validated all create, read, update, and delete (CRUD) operations with accurate edge and error conditions (Beck, 2004).

The JUnit test suite covers all core paths for each class. Boundary values were tested for field length constraints, and invalid scenarios were tested to ensure appropriate exceptions were thrown. For example, TaskTest.java includes tests for IDs that are exactly 10 characters, and names that are exactly 20 characters. Similarly, the AppointmentTest.java suite includes validations for null and past Date values, which meet the time-sensitive logic required by the application. These cases indicate the test suite is both comprehensive and practical. Manual inspection and test runner output confirmed a 100% test method success rate and covered all core logic blocks, meeting the rubric's 80% or higher test coverage requirement. The code was developed with a strong emphasis on technical correctness. Each setter in the classes like Contact, Task, and Appointment includes embedded validation logic, and tests were written to enforce and confirm these constraints. For instance, setPhone() in the Contact class was tested using a regular expression to ensure the value matched a 10-digit numeric format. Corresponding tests used lines such as assertThrows(IllegalArgumentException.class, () -> contact.setPhone("123")) to verify that any non-compliant data would trigger errors (Martin, 2008). Efficiency was considered in both implementation and testing. The use of utility methods in test files (e.g., getFutureDate() and getPastDate() in AppointmentTest) reduced code duplication and improved readability. Testing was limited to essential validations without over-testing known, trivial cases, reducing runtime and simplifying results interpretation (Myers et al., 2011). In TaskServiceTest.java, separate test cases for name and description updates kept the test logic concise and focused.

The primary testing strategy used in this project was unit testing with black-box and boundary value analysis. Tests were written based on requirements, not internal implementation, ensuring unbiased results (IEEE, 2012). Characteristics like field length, null checking, and value format were tested using edge values (e.g., max allowed characters). Exception testing was implemented using assertThrows to verify that constraint violations raised errors. Techniques such as integration testing, system testing, and mocking did not apply to this project scope, which focused solely on unit-level logic. These other techniques are beneficial for projects involving multi-module coordination, database interactions, or external APIs, but were outside the needs of this standalone, in-memory application. The methods used in this project are fundamental for validating individual components in any software system. Boundary testing and exception validation are valuable in ensuring data integrity (Myers et al., 2011). Integrating these techniques with mocks and stubs in more extensive systems can provide higher fidelity without needing a complete production environment. Projects involving microservices or user input-heavy systems would benefit from combining unit testing with integration and system testing for full coverage. A cautious mindset was maintained by strictly testing all validation logic and constraints per the rubric. For instance, the Appointment class required that dates not only be non-null but also in the future. AppointmentTest included logic that generated past dates with Calendar to implement this properly using add(Calendar.DAY\_OF\_YEAR, -1) to fail invalid scenarios explicitly. This approach avoided making assumptions about runtime conditions and enforced strict correctness. The code was split into multiple service and data classes to reduce bias, and tests were written independently to verify output based solely on input. This separation of concerns helped maintain objectivity, especially when validating invalid inputs. Using negative testing (intentionally passing bad inputs) helped ensure logic was not just confirming assumptions but truly catching edge cases and faults (Martin, 2008).

Discipline was shown by adhering to the rubric's rules without shortcuts. Fields were rigorously validated even when it seemed unlikely they would be misused. No hardcoded inputs or assumptions were made in tests, and all values used in assertions were generated or specified per case. In the future, this same discipline will be applied to avoid "technical debt" by building small, testable modules with accompanying unit tests and proper documentation. This reduces long-term risk and increases maintainability. The testing performed throughout Project One demonstrates the practical importance of structured unit testing. From field validations to service logic, each piece of functionality was rigorously tested to ensure compliance with project requirements and real-world robustness. Through a blend of caution, thoroughness, and strategic test coverage, the application is well-verified and maintainable. This experience has reinforced key software testing principles and established practices to guide future development efforts with quality and discipline (Beck, 2004; Martin, 2008; IEEE, 2012).

**References**

Beck, K. (2004). *Test-driven development: By example.* Addison-Wesley Prodessional. <https://www.pearson.com/en-us/subject-catalog/p/test-driven-development-by-example/P200000005570>

Gamma, E., & Beck, K. (1999). JUnit: A cook’s tour. *Java Report*, 4(5). <https://www.cs.cmu.edu/~pattis/15-1XX/common/zips/junit4.1/doc/cookstour/cookstour.htm>

Institute of Electrical and Electronics Engineers. (2012). *IEEE Standard for System and Software Verification and Validation* (IEEE Std 1012-2012). <https://standards.ieee.org/ieee/1012/4021/>​

JUnit Team. (n.d.). *JUnit 5 user guide*. <https://junit.org/junit5/docs/current/user-guide/>

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall. <https://www.pearson.com/en-us/subject-catalog/p/clean-code/P200000003416/9780132350884>

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). Wiley.