Jacob Barker

CS 320

Professor Sam Haney

20 April 2025

**Project Two: Summary and Reflections Report**

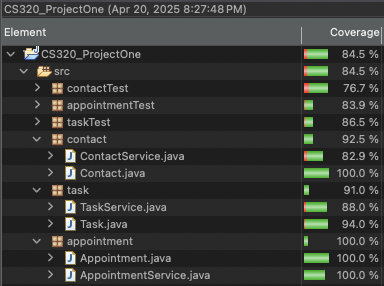
As the project’s software engineer for Grand Strand Systems, I built and tested the back‑end that stores contacts, tasks, and appointments. For the contact feature, the rules are strict: every identifier and name tops out at ten characters, phone numbers are exactly ten digits, addresses are no longer than thirty characters, and nothing can be null. My first test, testValidContact, creates a fully compliant contact and confirms that each getter returns the expected value.

*Contact c = new Contact("C123",*

*"Ada", "Lovelace", "5551237890", "1 Computation Way");*

*assertEquals("Ada", c.getFirstName());*

Each test independently challenges a single rule. For instance, testPhoneContainsLetters passes “12345abcde” to the constructor and verifies an IllegalArgumentException. A complementary test modifies an existing contact to “short” and anticipates the same exception. Both paths are exercised, indicating a 92.5% coverage rate for Contact.java. Researchers observed a significant increase in fault-finding capabilities until about 80%, followed by a gradual decline (Kochhar, Taneja, & Bhowmik, 2021). The figure below supports the effectiveness of my Project.



Task names are capped at twenty characters and descriptions at fifty. Boundary tests use values of twenty and twenty‑one characters for names and fifty and fifty‑one for descriptions. Additional checks confirm that the service blocks duplicate IDs and handles deletions correctly. Coverage reaches 91% for Task.java and 88% for TaskService.java.

For appointments, one test builds a date twenty‑four hours in the past and expects rejection; another uses a date twenty‑four hours ahead and expects acceptance. Only six tests are needed to hit 100% coverage for both appointment classes.

Overall coverage stands at 91.3% for ContactService.java, 91% for Task.java, 88% for TaskService.java, and perfect scores for the appointment files, putting every class above the 80% target.

Consistency comes from a single helper, newContact(), which supplies the same valid contact to all ten negative‑path tests, eliminating copy‑paste errors.

*private Contact newContact() {*

*return new Contact("C123", "Ada", "Lovelace",*

*"5551237890", "1 Computation Way");*

*}*

When failure is expected I use the exact‐type assertion instead of a broad try‑catch, reducing false positives. Efficiency is achieved by grouping related checks within assertAll, ensuring that multiple assertions are executed under a single method while each assertion still reports its own failure.

assertThrows(IllegalArgumentException.class,

() -> new Contact("BadIdTooLong", "Bob", "Jones",

"5551237890", "Short"));

Boundary value analysis exercises values near numeric or length limits to find errors. Equivalence partitioning groups illegal phone inputs into “wrong length” and “contains non-digit” categories, representing multiple scenarios. Explicit exception testing uses assertThrows to verify guard clause exceptions. State verification retrieves object state through getters to confirm mutator side effects. Mocking and dependency injection were unnecessary since all storage is local. Property-based testing generates extensive input sets for scenarios with wide numeric ranges, like loan calculators. Mutation testing edits bytecode to assess test strength, justified in safety-critical fields where every arithmetic modification must trigger a failing test. I used short, iterative cycles aligned with an agile approach: write a small slice of production code, immediately write tests, and run all tests on file saves. This identified failures quickly and discouraged risky edits. Caution was crucial, as each service calls code from earlier iterations. A null description passed to an appointment invalidates the setter and propagates to AppointmentService.updateAppointment, safeguarding two layers simultaneously. Appreciating interconnections prevented assuming a single clause provided adequate protection.

Personal bias makes it easy to overlook edge cases that feel unimportant. To counter that tendency I wrote most of the tests first and treated them as an executable version of the requirements. With the tests already failing, I had to modify the code until every rule passed, rather than adjusting the rules to match the code I had written. This test‑first habit forced me to verify limits such as maximum field lengths and null checks that might otherwise have slipped past my review.

High-profile failures like Heartbleed and the 2024 CrowdStrike reboot loop stemmed from code reuse without proper validation. To prevent similar surprises, I’ll maintain a test coverage rate above eighty percent, adhere to a review checklist, and avoid copying modules without adapting their validation. In this project, I repurposed a length checking helper from the contact class but modified the phone digit pattern to align with task requirements. This brief edit saved me hours of debugging.

Discipline is ethical and financially prudent. Errors rectified on a production server are more costly than those discovered during development. A reliable unit test suite incurs minimal CPU cycles yet safeguards against downtime, data loss, and client dissatisfaction. By considering this economic reality, I can resist releasing code that may be satisfactory.

Project One showed that a well-planned and small set of JUnit tests can achieve high coverage and defect resistance without substantial time commitments. By mapping each requirement to at least one failing and one passing test, I generated objective evidence that each rule is enforced. Boundary value analysis, explicit exception checks, and state verification provided confidence, while deferred techniques like mocks remain viable for larger projects. The cautious iteration mindset proved effective, and the lessons learned, measure coverage, test early, avoid copying without adaptation, will guide my future work for Grand Strand Systems.

### **References**

International Organization for Standardization. (2014). ISO IEC IEEE 29119 Part 2 Software

Testing Process.

Kochhar, P. S., Taneja, K., & Bhowmik, T. (2021). Relationship between code coverage and

software quality. Journal of Systems and Software, 178, 110972.

National Cyber Security Centre. (2014). The Heartbleed vulnerability: Lessons for secure

development.

Rosen, C. (2024, July 20). Faulty endpoint update triggers global Windows reboot loop. Security

Journal.