Software Testing Portfolio - B244668 (GitHub Repository)

Software Being Tested

The software under test is a drone-based medication delivery system from my Informatics Large Practical project, which optimizes deliveries via autonomous drones and exposes REST APIs for distance, position, region checks, restricted-area pathfinding, drone queries, and route comparison.

The full repository is available here. It has the source code, an evidence document with easily accessible screenshots of relevant source code, and each learning outcome documented in more depth than this portfolio can cover.

Learning Outcomes

1. Analyze Requirements to Determine Appropriate Testing Strategies (25%)

1.1 Range of Requirements: The project addressed a broad range of functional and non-functional requirements. Functional requirements included delivery route planning, path validation, error handling, and scheduling across multiple drones. Non-functional requirements addressed robustness (e.g., how the system handles malformed inputs), correctness (returning optimal paths), and responsiveness (although performance testing was deprioritized due to time). These requirements were based on the ILP project specification and refined during implementation planning. Figure 1.1 in the Evidence Document highlights the initial mapping of requirements to testing strategies.
1.2 Requirement Levels (System, Integration, Unit): Testing was conducted across multiple abstraction levels. Unit tests focused on isolated services like `PathfindingService`, `RegionService`, `ValidationService`, and `DroneAvailabilityService`. Integration tests validated the interaction between components, especially the API controller and its dependencies. System-level testing involved simulating end-to-end request scenarios to verify delivery behavior under realistic conditions. This layered approach ensured that tests targeted both granular correctness and overall functionality. Figures 1.3 and 1.5 demonstrate unit and integration testing examples, respectively.
1.3 Test Approaches for Key Attributes: Each requirement was matched with a fitting testing approach. For numeric inputs like distance and capacity (payload limits), boundary value analysis was used. Qualitative attributes such as robustness were addressed with negative test cases that passed invalid or incomplete data to the API. Functional behaviors were tested using integration and scenario-based testing. Parameterized tests supported efficient test coverage of varied inputs. The chosen approaches ensured representative coverage of key system behaviors and quality concerns. Figures 1.6 and 1.9 show representative tests, including checks against the `/calcDeliveryPath` endpoint.
1.4 Assessing Testing Approaches: Given time constraints, the test strategy was appropriate and focused on high-value targets. Critical services and API endpoints were thoroughly tested using a mix of blackbox and white-box techniques. While some quality attributes like performance were not exhaustively tested, the chosen methods achieved adequate functional confidence and a high degree of robustness. The test suite includes 143 tests (all passing), and the strategy could be extended in future iterations to include more extensive fault injection or load testing.

Refer to this pdf for LO1 detailed evidence and explanation.

2. Design and Implement Comprehensive Test Plans with Instrumented Code (25%)

2.1 Test Plan Overview: A test plan was developed early, aligning each software requirement with corresponding test classes and techniques. It included plans for unit tests of algorithmic modules, integration tests for service interactions, and error-case tests for API validation. Requirements were prioritized by their criticality to delivery

safety and functionality. The plan was iteratively updated as the implementation evolved, ensuring relevance. An overview is presented in Figure 2.1.
2.2 Test Plan Quality: The test plan proved robust in coverage and execution. All major modules were included, and the plan was effective in guiding development-stage testing. However, certain test types like performance and security analysis were excluded to remain within project scope. The quality of the plan was validated through traceability to coverage and test results, as shown in the Evidence Document. Future improvements could include automated traceability metrics and expanded test matrices.
2.3 Code Instrumentation:The project employed JaCoCo to instrument code and collect coverage metrics. This was integrated into the Maven build lifecycle, providing automatic coverage reports after each test run. Spring Boot's testing framework and MockMvc facilitated realistic HTTP test scenarios without full deployment. Mockito allowed control over dependencies for focused unit tests. Instrumentation was non-intrusive and had minimal overhead during local test execution.
2.4 Instrumentation Evaluation: The instrumentation was reliable and allowed for iterative improvements to the test suite. Coverage reports revealed under-tested branches and classes, helping target new tests. For example, low branch coverage in region validation prompted new boundary tests. Figures 2.3 and 3.6 show how instrumentation directly informed test refinement. This feedback loop was essential in validating the effectiveness of the test plan and guiding improvements.

Refer to this pdf for LO2 detailed evidence and explanation.

3. Apply a Wide Variety of Testing Techniques and Compute Test Coverage and Yield According to a Variety of Criteria (20%)

3.1 Range of Testing Techniques: A wide range of testing techniques was employed. These included equivalence partitioning for input validation (e.g., valid vs invalid coordinates), boundary value analysis for numeric limits (capacity, path length), and parameterized testing to exercise combinations of request parameters. Mocking isolated services in unit tests, while integration tests verified full request processing. This blend enabled targeted and realistic testing. It also ensured that core logic and external interfaces were evaluated under diverse conditions.
3.2 Test Adequacy Criteria: Adequacy was assessed using both quantitative and qualitative criteria. Instruction and branch coverage from JaCoCo gave insight into structural completeness, while requirement traceability checked that each core behavior was tested. The variety of inputs and negative cases also contributed to confidence in adequacy. Achieving 60% instruction and 38% branch coverage, while short of initial targets, was sufficient given time constraints. Future adequacy evaluations could benefit from mutation testing or decision condition coverage.
3.3 Testing Results: The final test suite included 143 tests (unit + integration), all passing (Figure 4.2). Tests uncovered and led to correction of several logic flaws (e.g. drone unavailability not propagating to scheduler). Test yield was high during iterative development, and coverage metrics improved consistently through feedback-driven refinement. A complete breakdown of test classes and outcomes is logged in the repository.
3.4 Evaluation of Results: Defect resolution via test feedback demonstrated utility. Tests provided confidence in controller-service behavior. Remaining gaps (e.g. performance metrics) were known and justified due to scope. In terms of yield, the tests were effective in catching both minor bugs and structural flaws. Limitations were tracked and documented for future improvements, and plans for extended testing phases post-course were outlined.

Refer to this pdf for LO3 detailed evidence and explanation.

4. Evaluate the Limitations of a Given Testing Process, Using Statistical Methods Where Appropriate, and Summarise Outcomes (15%)

4.1 Gaps in Testing Process: The testing process had known limitations. Performance, concurrency, and security testing were out of scope. Certain branches related to rare edge cases remained untested due to difficulty

reproducing conditions. These were logged in test coverage reports for transparency. In future iterations, more extensive use of fault simulation and automated state exploration could help mitigate these gaps.
4.2 Target Coverage/Performance: Initial goals aimed for 70%+ instruction, 50%+ branch. These figures were selected based on expectations of comprehensive test depth while accounting for project complexity and resource limitations. JaCoCo reporting helped guide real-time tracking toward those targets.
4.3 Comparing Testing to Targets: Final coverage was approximately 60% for instructions and 38% for branches. This shortfall was analyzed, and the most significant missed paths involved unlikely or defensive checks (Figure 4.1). While the exact targets weren't met, the quality and strategic prioritization of tested logic provided strong assurances of system reliability.
4.4 Steps to Reach Target Levels: To reach the original coverage goals, additional effort would be needed in mocking rare scenarios, simulating backend failures, and applying mutation testing. Performance testing would require synthetic load tests and profiling tools. These extensions were noted for post-course exploration. In particular, concurrency and stress tests could be added for better real-world resilience assessment.

Refer to this pdf for LO4 detailed evidence and explanation.

5. Conduct Reviews, Inspections, and Design and Implement Automated Testing Processes (15%)

5.1 Review Criteria and Code Issues: Manual and peer reviews were conducted on the pathfinding and controller logic. A peer identified a missing case in drone unavailability that was later resolved via added tests (Figure 1.9). IntelliJ IDEA static analysis inspections supported code health and maintainability. While Checkstyle or SpotBugs were not configured, IDE inspections helped enforce style consistency. These reviews helped enforce both correctness and style consistency, and they reduced technical debt.
5.2 CI Pipeline Construction: CI pipeline was implemented using Maven's build lifecycle, executed locally via `mvn clean verify`. The process includes compilation, test execution via Surefire, and JaCoCo coverage reporting. The workflow was designed to be lightweight, modular, and easily extensible, supporting testing as well as documentation generation. While GitHub Actions was not configured, the Maven-based process follows CI principles and can be extended to remote execution.
5.3 Test Automation: Test execution and coverage reporting fully automated (Figure 5.2). JaCoCo output was generated automatically in HTML format in `target/site/jacoco/` after each test run. This automation ensured consistent execution and reduced overhead in verifying outcomes after each code change.
5.4 CI Pipeline Evaluation: Maven-based CI pipeline successfully validated the software's build integrity and testing coverage at every development milestone. All 143 tests consistently passed across modules, and no test regressions were observed during iterative development. Coverage results (~60% instruction, ~38% branch) were updated automatically after test runs. Local HTML reports enabled review of test gaps without requiring external hosting or workflows. The pipeline helped enforce discipline across test execution and allowed quick feedback during code integration.

Refer to this pdf for LO5 detailed evidence and explanation.

This document serves as a concise self-evaluation of how the software testing learning outcomes have been achieved in practice. Supporting documents with in-depth evidence for each learning outcome are available in the repository.