Software Testing 2025/6 Portfolio

Zoran Ma

January 18, 2026

1 Outline of the Software Being Tested

For this coursework, I am testing my ILP project called **PizzaDrone**. It is a Java Spring Boot application designed to help drones deliver pizzas to students in Appleton Tower. The system has two main jobs: (1) it validates orders to make sure they follow business rules (like checking if the restaurant is open), and (2) it efficiently calculates the shortest flight path for the drone, making sure to avoid "No-Fly Zones".

2 Learning Outcomes

1. **Analyze requirements to determine appropriate testing strategies** [default 20%]

   (a) **Range of requirements, functional requirements, measurable quality attributes, qualitative requirements**
   Full details are in `docs/requirements.md`. I derived requirements by considering **Stakeholders** (e.g., Students, University Admin, Drone Operators). **Functional:** I distinguished between **Correctness properties** (ensuring order logic produces correct results, FR1–FR6) and **Safety properties** (ensuring the system *cannot* violate safety rules like entering No-Fly Zones, FR9). **Qualitative (QR1–QR4):** Ensured the system is data-driven (fetching external APIs), robust against malformed JSON, and meets the 60s performance target.

   (b) **Level of requirements, system, integration, unit**
   As detailed within `requirements.md`, I structured requirements to be tested at three distinct levels. I targeted the Unit Level for isolating complex algorithmic logic (e.g., distance checks) and strict validation rules (FR1–FR6). I addressed the Integration Level to verify that the `OrderService` correctly orchestrates internal logic with data fetched from external providers (QR1). Also, I covered the System Level by treating the application as a black box to verify the REST API contract and HTTP status codes (FR12–FR14).

   (c) **Identifying test approach for chosen attributes** To verify the attributes mapped in `requirements.md`. I used *JUnit 5* tests to efficiently cover the edge cases of validation rules (FR1-FR6). To handle the dependency on the external ILP service (QR1), I used *Mockito* to simulate external responses. This ensures test determinism and allows simulation of "dirty data" without relying on the unstable live environment. For API verification, I utilized *MockMvc* for HTTP testing, while performance constraints (QR4) were enforced using strictly defined JUnit timeouts.

   (d) **Assess the appropriateness of your chosen testing approach** This strategy effectively maximizes Verification given the project constraints. Mocking is appropriate to avoid "flaky tests" caused by external service instability. However, a key limitation is the difficulty of performing validation under realistic conditions. My approach relies on mocks rather than live data, which means I cannot fully validate system behaviour against real-world network latency or silent API schema changes.

2. **Design and implement comprehensive test plans with instrumented code** [default 20%]

(a) **Construction of the test plan**

A test plan is detailed in `docs/test_planning.md`. It follows an XP-style workflow and a DevOps-style verification phase, where unit and API/integration tests are executed automatically via CI on each push.

(b) **Evaluation of the quality of the test plan**

The evaluation in `docs/test_planning.md` prioritises testing by safety and business risk, and identifies vulnerabilities such as limited realism when external services are mocked and reduced confidence in live end-to-end behaviour. Risks (schedule/technology) and mitigations (early scaffolding, incremental test-first development) are documented.

(c) **Instrumentation of the code**

Instrumentation and scaffolding are defined in `docs/test_planning.md`. I implemented **SLF4J logging** in `OrderService` to expose key validation decision points without leaking sensitive data. For scaffolding, **Mockito** isolates unstable external API dependencies and **MockMvc** drives REST endpoint tests.

(d) **Evaluation of the instrumentation**

The instrumentation is effective. The targeted log messages allow me to diagnose logical flows and clearly distinguish what has failed. Regarding scaffolding, while Mocking improved the determinism of my tests, it may diverge from live conditions. Therefore, the timing checks are treated primarily to prevent infinite loops, rather than testing performance under extreme environments.

3. **Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria** [default 20%]

(a) **Range of techniques**

I implemented a mixed strategy combining systematic functional testing, structural testing, and API-level integration checks. Specification-based tests apply equivalence partitioning and boundary value analysis to key order-validation rules (FR1–FR6). Structural isolation is achieved using Mockito for external dependencies (QR1), and controller/API behaviour is verified with MockMvc to check correct HTTP outcomes (FR12–FR14). Path calculation is additionally exercised by service-level tests that assert required correctness properties (e.g., avoiding no-fly zones and respecting the move limit).

(b) **Evaluation criteria for the adequacy of the testing**

Adequacy is assessed using coverage over the chosen requirements in `docs/requirements.md`. Also, structural coverage (line/method coverage from IntelliJ), and number of failures detected during development and regression stability in CI.

(c) **Results of testing** The executed test suite contains 59 tests with a 100% pass rate. Coverage results show 93% line coverage and 92% method coverage overall (see `docs/test_results.md` and the coverage screenshot).

(d) **Evaluation of the results** The evaluation confirms that the test suite is robust for internal logic, with high coverage. However, most integration is verified under mocked external data, so behaviour under live ILP service changes and real network conditions is not fully validated.

4. **Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes** [default 20%]

(a) **Identifying gaps and omissions in the testing process**

The current tests provide strong unit and controller-level confidence, but there are clear omissions. First, most external dependencies are mocked, so the tests do not fully cover realistic failures such as network latency. Second, geometric safety requirements (e.g., No-Fly Zones) are mainly validated on returned path points, boundary cases (points on polygon edges, floating-point tolerance) are not systematically tested. Third, the performance check is treated as a smoke regression only, and CI timing variance means it cannot certify the 60s requirement under all conditions.

(b) **Identifying target coverage/performance levels for the different testing procedures**

For FR1–FR6 and FR12–FR14, a target is full specification coverage at the rule level: each rule has at least one positive and one negative test, with boundary values for numerical constraints. For structural adequacy, a target is $\geq 90\%$ line coverage in service/util level to ensure the core algorithm is thoroughly verified. For the measurable attribute (QR4), the target is a repeatable regression check: representative path calculations should complete well below the 60s limit, while reporting timing statistics to monitor trends.

(c) **Discussing how the testing carried out compares with the target levels**

The achieved structural coverage is high (as shown in the coverage report), and key business rules and controller contracts are covered with passing tests. However, the tests fall short of the above targets like realistic end-to-end behaviour against the live ILP service, systematic boundary exploration for geometric constraints, and statistically meaningful performance measurement.

(d) **Discussion of what would be necessary to achieve the target level**

To close these gaps, I would add a small end-to-end test stage against the live ILP endpoints using a fixed set of representative orders, introduce invariant-based checks over every produced flight path (step size, no-fly exclusion, central-area monotonicity, move limit), and strengthen performance evaluation by repeated runs on fixed inputs and reporting mean or variance to distinguish genuine regressions from CI noise.

5. **Conduct reviews, inspections, and design and implement automated testing processes** [default 20%]

(a) **Identify and apply review criteria to selected parts of the code and identify issues in the code**

During code review, I used simple criteria: readability, basic input checking, and clear error handling. For readability, I found some variable names that are too short (e.g., single-letter names), which makes the code harder to understand. For robustness, I noticed that some service methods assume the input is always correct. If the controller misses a bad input, this could cause unexpected behaviour. For error handling, I looked for places where exceptions are handled too broadly, which can hide the real cause of a failure. I also noticed small style issues, such as mixed use of `Double` and `double` in the *LngLat* model. While this does not affect functionality in the current implementation, it can introduce unnecessary risk (e.g. null handling) and reduces code consistency and readability.

(b) **Construct an appropriate CI pipeline for the software**

I implemented a CI pipeline using GitHub Actions (see `.github/workflows/ci.yml`). The current pipeline executes the Maven lifecycle on every push. The **Build stage** ensures successful compilation, and the **Test stage** executes the full JUnit test suite.

As a planned extension, I designed a **Quality Gate** stage using `jacoco:check`. This stage would enforce a minimum coverage threshold (e.g. 90%) and prevent merges that reduce test adequacy.

(c) **Automate some aspects of the testing**

Automated testing is integrated into the repository through the CI workflow. Each push automatically triggers unit and controller tests, providing fast feedback. Only merge to the main branch when CI checks passed.

(d) **Demonstrate the CI pipeline functions as expected**

The CI pipeline demonstrates correct behaviour by catching different defect classes at different stages: compilation errors during build, functional bugs during test execution, and potential quality degradation through the planned coverage gate. Evidence of successful and failing CI runs is visible in the GitHub Actions history.