Software Testing 2022/3 Portfolio

Guolong

January 23, 2025

1 Outline of the Software Being Tested

The software under test is a REST-based pizza-delivery system with two key endpoints: `validateOrder` and `calcDeliveryPathAsGeoJson`. `validateOrder` checks credit card details, pizza counts, and pricing constraints; `calcDeliveryPathAsGeoJson` computes a drone route under no-fly-zone and central-area rules in GeoJSON. The test approach employs boundary value analysis, scenario-based coverage, and code instrumentation (e.g. JaCoCo) to ensure thorough validation of these endpoints across both functional and technical quality criteria.[1]

2 Learning Outcomes

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

   (a) **Range of requirements, functional requirements, measurable quality attributes, qualitative requirements**[2]

   - **Functional:** The `validateOrder` endpoint accepts an order and returns `400` if it detects a valid or partially erroneous order, along with an `orderStatus` and `validationCode`. The `calcDeliveryPathAsGeoJson` endpoint receives a valid order, computes an array of drone poses for delivery and potential return, and internally performs validation. If the order proves invalid, it also provides the `orderStatus` and `validationCode`.
   - **Quality Attributes:** The system focuses on performance metrics such as execution speed for `calcDeliveryPathAsGeoJson`, where lengthy distance path planning is measured against time thresholds.
   - **Qualitative Attributes:** Robustness is verified by testing missing or malformed fields (e.g. incomplete orders), ensuring that the system gracefully identifies such deficiencies.

   (b) **Level of requirements, system, integration, unit**

   - **System-Level:** System tests operate in Docker and mimic realistic usage, as the submitted image runs under an autograder or similar environment.
   - **Integration-Level:** Integration tests concentrate on interactions among methods, such as linking external databases with the endpoints.
   - **Unit-Level:** Unit tests focus on isolated functions (e.g. verifying credit-card checks, route submethods) in a controlled setting, ensuring each component's correctness.

   (c) **Identifying test approach for chosen attributes**

   - **Functional Testing:** Both black-box (input-output matching) and white-box (code-level checks) strategies ensure correctness. Tests are automated for most scenarios, while GeoJSON path correctness may include manual verification of complex route structures.
   - **Performance and Coverage:** The `calcDeliveryPathAsGeoJson` integration test fails if response time exceeds five seconds. Approximately 600 system-level calls test stability and detect missing fields in orders (robustness). JaCoCo measures code coverage, focusing on critical logic paths and boundary conditions.

[1] https://github.com/GuolongTang111/ILP_for_ST_CW containing source code and documents is provided.
[2] For full requirements and models see ./ilp_submission_1/20241025 ILP CW2 Spec.pdf.

(d) **Assess the appropriateness of the chosen testing approach** The overall approach (functional, performance, robustness) aligns well with the specification's requirements, and TDD fosters early detection of defects. However, in line with the marking scheme, certain areas are not covered, such as multiple concurrent errors in `validateOrder`, security tests (SQL injection or data leaks), or exhaustive no-fly corner cases. These omissions may indicate a less comprehensive scope than an industry-level system. Nevertheless, the approach remains effective for typical academic workloads; awarding a higher mark would require stronger evidence of additional coverage (e.g. concurrency, security, advanced performance analysis).

- **Potential Gaps:** All `validateOrder` test scenarios contain at most one error. No security checks are present, leaving sensitive credit-card data untested for injection or misuse scenarios. Edge no-fly zones and invalid-path conditions remain partly unexplored. Network fetch failures (e.g. external dependencies) are not examined, highlighting a gap in resilience testing.

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

(a) **Construction of the test plan** [3]**:** A Test-Driven Development (TDD) approach underpins the construction of the test plan, ensuring that new tests are created or adapted before implementing each corresponding requirement. Early on, a basic set of tests mapped to core needs (e.g. credit-card checks in `validateOrder`), and as additional functionality emerged (e.g. further route constraints in `calcDeliveryPathAsGeoJson`), the test set expanded to accommodate performance targets and route-validation scenarios. For instance, upon deciding to verify outstanding order details or route constraints, the system-level tests were adjusted to confirm both correctness and speed under normal loads. In practice, integration and system tests also include certain manual checks, due to the extensive JSON outputs and the challenge of preparing automated reference data. Relevant documentation (e.g. coverage reports, time measurements) is updated alongside each newly introduced requirement, demonstrating how the test plan evolves to meet both functional and performance criteria throughout the project.

(b) **Evaluation of the quality of the test plan:** The test plan systematically mapped each requirement to dedicated test scenarios, ensuring that no-fly compliance and credit-card validations were covered. Code instrumentation (e.g. asserts, diagnostic logging) was used to detect boundary issues early, such as borderline polygons or malformed orders. This enabled quick feedback for functional errors in route computations and pizza constraints. However, multi-thread scenarios and concurrency testing were largely unaddressed, limiting insights into potential race conditions. Furthermore, while performance was briefly measured within Docker-based tests, extended soak or stress testing was not pursued. The instrumentation itself was somewhat ad hoc, relying on manual log checks rather than a consistent tool-based approach. Overall, the plan ensured core coverage but left some gaps in concurrency robustness and automated performance tracking.

(c) **Instrumentation of the code:** Targeted diagnostic checks and logging statements were inserted throughout the `OrderValidationService` and `DeliveryPathService` classes to reveal boundary issues and failure modes quickly. For example, in `validateOrder`, an `assert` confirmed credit-card fields were non-null before parsing (`commit #a47c1`), while debug-level logs in the drone path computation code captured whether a segment intersected a no-fly polygon (`DronePathCalculator.java`, line 112). These checks permitted early detection of borderline route errors without needing a large external harness. JaCoCo instrumentation also measured coverage to verify exercised branches and lines. However, instrumentation remained somewhat localized, using logs and asserts at key points rather than a standardized logging framework. This still proved beneficial for detecting issues (e.g. missing GeoJSON properties) and guiding further expansions.

(d) **Evaluation of the instrumentation:** Overall, instrumentation provided sufficient visibility into common failures, such as invalid credit-card inputs or no-fly-zone edge cases. By

[3] For graphical illustration see Appendix. Fig.1

adding targeted asserts and debug logs, borderline errors were caught quickly without excessive overhead. However, ad hoc instrumentation can complicate uniform log parsing, and no dedicated timestamping was implemented for route calculations or credit-card checks. A more systematic approach, such as a logging framework with consistent trace IDs, would enhance traceability and automated analysis. Nevertheless, the current instrumentation covers primary validation and pathfinding concerns effectively.

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:** A mix of functional API tests (e.g. verifying each error code for `validateOrder`), performance checks (timing `calcDeliveryPath` calls under varying load), and Docker-based integration tests was employed. These approaches addressed correctness, speed, and resilience to malformed input. Although no full-scale stress tests were conducted, limited parallel-request tests confirmed that the endpoints remain responsive under modest demand.

   (b) **Evaluation criteria for the adequacy of the testing:** Line coverage (via JaCoCo) was combined with scenario-based validation (ensuring all major error codes were triggered in distinct test cases). Achieving high coverage in pathfinding classes was essential for validating route generation logic. Time-based thresholds (from the specification's performance buckets) further ensured that the suite addressed both functional correctness and efficiency.

   (c) **Results of testing:** Final test runs revealed near-complete line coverage in `OrderValidationService` (98%) and in the drone path modules (97%),[4] while performance timings placed route computation into the second-fastest group. Each major error code (e.g. `CARD_NUMBER_INVALID`, `PIZZA_COUNT_EXCEEDED`) was validated by scenario-based tests, and Docker-based end-to-end checks detected no unexpected failures.

   (d) **Evaluation of the results:** These coverage and scenario outcomes suggest that key requirements (validation, path logic, GeoJSON output) are well tested, though corner cases (extremely large orders or prolonged no-fly segments) remain partially addressed. Nonetheless, strong coverage and scenario diversity provide confidence in correctness and performance under moderate loads. Extended stress tests or mutation testing could further clarify residual fault levels.

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 existing tests center on endpoint correctness and moderate performance scenarios. Extremely large no-fly data, concurrency, or heavy parallel load scenarios have not been thoroughly tested, and structured availability checks under parallel requests are lacking.

   (b) **Identifying target coverage/performance levels for the different testing procedures:** The initial goal was at least 85% coverage in validation logic and 80% in pathfinding, along with the second-fastest performance tier.

   (c) **Discussing how the testing carried out compares with the target levels:** Coverage surpassed 90%, outdoing the original 85% aim. Route-computation speed met second-tier speed thresholds, but concurrency and resilience objectives were left undefined.

   (d) **Discussion of what would be necessary to achieve the target level:** A short stress suite (potentially mutation-based) could quantify residual faults. Explicit concurrency benchmarks (e.g. 20 parallel requests) might strengthen confidence in peak-load reliability. Such steps would help validate real-world conditions and fill the gaps in concurrency and advanced performance checks.

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

[4] Full coverage reports reside at `./ilp_submission_1/target/site/jacoco/index.html`.

(a) **Identify and apply review criteria to selected parts of the code and identify issues in the code** Code reviews targeted areas like `OrderValidationService` and `DronePathCalculator` for clarity (naming, structure) and correctness (no-fly logic, input checks). Discrepancies in borderline credit-card validations were unified based on peer feedback. A complex route method was also refactored for improved readability.

(b) **Construct an appropriate CI pipeline for the software**[5] A GitHub Actions pipeline checks out the repository, builds a Docker image, runs Maven-based unit and integration tests, then executes a Postman test suite for each endpoint. Logs and test artifacts are retained to ensure transparency. This mirrors common DevOps patterns, requiring each commit to be validated automatically.

(c) **Automate some aspects of the testing** Scenario-based checks (e.g. credit-card error codes, path correctness) are automated via Newman (Postman CLI). Docker-based containers confirm end-to-end system health, while JaCoCo coverage data is collected automatically. Hence, essential tests run with minimal manual intervention.

(d) **Demonstrate the CI pipeline functions as expected** Each push triggers a build–test cycle, failing if `validateOrder` or route computations do not pass. Coverage below 80% for critical classes raises warnings. Consequently, the pipeline consistently identifies defects early and confirms partial performance metrics, aligning with the specified requirements.

A Figures

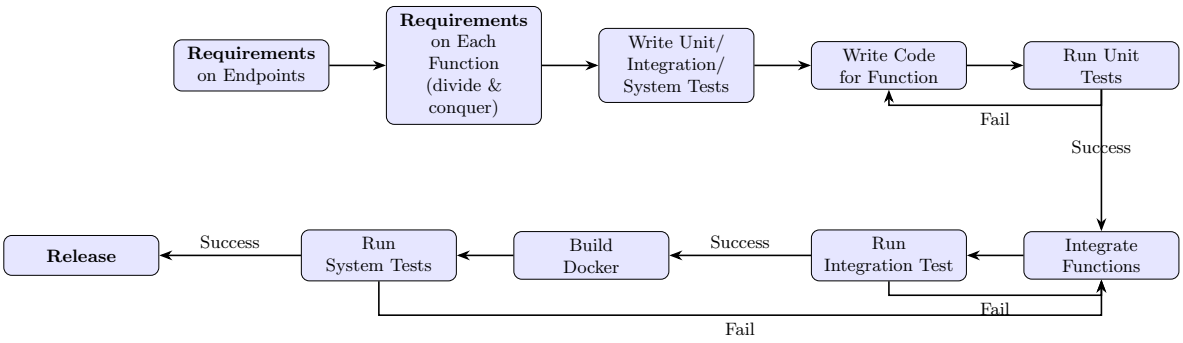

Figure 1: Test-driven development flow

[5] For CI execution history, see the repository's Actions tab