Learning Outcome 2 – Design and Implement Comprehensive Test Plans with Instrumented Code

2.1 Detailed Explanation of Tests Performed and Construction of the Test Plan

The test plan for the ILP drone delivery project was designed to ensure functional correctness, resilience, and integration of the system's services. These include PathfindingService, DroneAvailabilityService, RegionService, ValidationService, and the higher-level delivery orchestration provided through DeliveryPathService. The plan was created incrementally as the system evolved, aligning with test-driven principles and requirement traceability.

Unit Tests

Unit tests were created for critical service classes. For example:

- **A Pathfinding*:** Tests in PathfindingServiceTest validate A* algorithm behavior, such as proper avoidance of no-fly zones, null/empty result handling, and safe behavior near zone boundaries. Edge conditions were tested explicitly using polygonal region data (e.g., calculatePath_aroundRestriction, calculatePath_restrictedAreaBlocksStartOrEnd).
- **Region Containment:** RegionServiceTest verified accurate point-in-polygon logic, including edge cases for points inside, outside, on boundaries, and invalid inputs. Tests ensured that delivery paths avoided restricted areas and responded accurately at region boundaries. GeoJSON parsing and structure validation were tested in integration tests for endpoints like /calcDeliveryPathAsGeoJson.
- **Drone Filtering:** DroneAvailabilityServiceTest checked drone filtering logic by battery, availability, and cooling status, with edge-case tests on battery thresholds and move limits.

Integration Tests

Integration tests were written using Spring Boot's MockMvc for endpoints like:

- **/calcDeliveryPath:** Validated full-stack behavior, ensuring proper chaining of validation, region checks, pathfinding, and delivery logic.
- **/queryAvailableDrones and /droneDetails/{id}:** Verified filtering behavior and correct status responses.
- **/isInRegion:** Tested various region containment queries and malformed requests.
- These tests provided input diversity (malformed, edge, valid) and verified JSON outputs, HTTP status codes, and error handling.

System Tests

End-to-end system tests simulated full delivery cycles. Parameterized test data (e.g., delivery orders, customer positions) validated that dispatch sequences triggered correct route planning and error fallback logic.

Robustness and Negative Testing
Invalid data cases were tested across endpoints: malformed GeoJSON, null fields, invalid coordinates, unsupported drone states. All endpoints were tested for 4xx error propagation and validation logic was verified with simulated failure conditions.

2.2 Evaluation of the Quality of the Test Plan

The test plan demonstrated broad and deep coverage of the system:

- **Requirement Traceability:** All public APIs had associated test classes and linked to underlying service methods.
- **Edge-Case Handling**: Specific tests validated behavior on boundary coordinates, minimum battery thresholds, and malformed region polygons.
- **Exploratory Gaps**: Early bugs (e.g., polygon boundary misclassification) led to targeted test additions. Drone filtering logic was refined after tests exposed edge battery-level errors.

The final plan evolved across development milestones and reflected realistic deployment concerns. Areas such as performance and security were acknowledged but deprioritized due to scope and time limits.

2.3 Instrumentation of the Code

Code was instrumented using JaCoCo integrated through the Maven build lifecycle. This generated HTML reports displaying:

- **Instruction Coverage:** Code execution paths at the bytecode level
- **Branch Coverage:** Conditional logic branches executed per test run

Spring's MockMvc was executed with instrumentation enabled, allowing coverage analysis across service and controller layers.

Instrumentation revealed missed logic in areas like complex region validation conditions, complex availability filters, and rare error handlers. These insights directly informed additional unit and integration tests.

Mockito was used to stub service dependencies, enabling precise control and verification of method calls in logic-heavy services like DroneAvailabilityService.

2.4 Evaluation of the Instrumentation

JaCoCo instrumentation reported approximately 60% instruction coverage and 38% branch coverage. This level was acceptable given the complexity of service interdependencies and error handling layers.

Notable insights:

- **Low-Coverage Areas:** Defensive code paths (e.g., null-checks, bad input guards) in ValidationService remained under-tested.
- **Coverage Feedback Loop:** Instrumentation enabled a test-refinement loop, especially for poorly tested edge-case flows.

While additional instrumentation (e.g., mutation testing, performance timers) could have improved analysis, JaCoCo provided the necessary structural overview. Reports were regularly reviewed and used to guide further test development. All 143 tests were executed with instrumentation enabled.

This document provides detailed reasoning behind the test design, coverage-driven test expansion, and instrumentation methods used in the ILP drone system. References to key service tests, coverage goals, and real examples (e.g., A* behavior, no-fly zone logic) support the comprehensive scope of the testing plan. Supporting screenshots and coverage reports are available in the GitHub repository and Evidence Document.