Learning Outcome 1 – Analyze Requirements to Determine Appropriate Testing Strategies

1.1 Range of Requirements

The ILP drone delivery software met a blend of functional and non-functional requirements. Functional requirements included REST API endpoints such as /calcDeliveryPath for route generation, /droneDetails/{id} for individual drone queries, /query and /queryAvailableDrones for drone filtering, and /isInRegion for region validation. Each of these required robust and consistent behavior across varying inputs.

Non-functional requirements included:

- **Robustness:** The system had to handle malformed inputs, invalid coordinates, missing delivery data, and absent drone states without crashing.
- **Performance:** Though performance testing was not fully scoped, efficient path calculation and quick filtering were expected for responsive user interaction.
- **Correctness:** Results had to conform to geographic constraints (e.g., avoiding no-fly zones) and return precise and valid paths or errors.

Some requirements were implicit in module behavior, such as greedy TSP path optimization in DeliveryPathService and battery/move budgeting via maxMoves fields. The combination of explicit RESTful contracts and internal logic boundaries informed the testing criteria.

1.2 Level of Requirements

Testing spanned three levels:

- **System Testing:** End-to-end behavior was verified through MockMvc tests of endpoints like /calcDeliveryPath, simulating full request-response cycles.
- **Integration Testing:** Interactions between services such as PathfindingService, RegionService, and ValidationService were validated to ensure correct flow through internal calls.
- **Unit Testing:** Individual services were isolated and tested for logical correctness. For example, RegionService was tested for its polygon containment logic and boundary-edge handling, while DroneAvailabilityService was verified against availability and battery conditions.

This tiered testing ensured both low-level correctness and high-level orchestration fidelity.

1.3 Identifying Test Approach for Chosen Attributes

The test strategy incorporated several structured approaches:

- **Equivalence Partitioning:** Applied to coordinate ranges (valid vs. out-of-bounds), drone battery levels, and delivery payloads.

- **Boundary Value Analysis:** Targeted fields like maxMoves (range capacity) and payload thresholds (dispatch capacity).
- **Negative Testing:** Invalid JSON payloads, null fields, and requests missing mandatory attributes were submitted to confirm safe rejections.
- **Exploratory Testing:** Focused on realistic but unexpected inputs, such as empty region maps or drones with borderline energy.
- **Robustness Testing:** Included malformed region data and delivery requests to ensure validation layers could gracefully handle and reject incorrect data.

The test suite comprised 143 total tests, distributed across unit, integration, and controller-level paths, all automated using JUnit 5 and Spring's MockMvc.

1.4 Assessing the Appropriateness of the Testing Approach

The chosen testing strategy was well-suited for the safety-critical, modular nature of the drone delivery system. Unit tests allowed tight control and logic verification; integration tests ensured coherent behavior across services; and system tests validated full-stack operations and error resilience.

Coverage reports using JaCoCo revealed approximately 60% instruction coverage and 38% branch coverage. While not exhaustive, these metrics indicated meaningful reach into key service paths. Most uncovered branches existed in complex or defensive logic blocks (e.g., complex boolean conditions in RegionService).

Although performance and concurrency testing were not conducted due to project scope, the combination of structured test design, high test count, and clear traceability to requirements affirmed the overall appropriateness of the testing approach.

This document presents a detailed mapping between system requirements and applied testing strategies. The inclusion of layered testing levels, structured test design techniques, and automated instrumentation ensures the software's resilience and correctness. Evidence and supporting figures are available in the GitHub repository and Evidence Document.