Test Planning

Construction of the Test Plan

The Test Plan was developed based off the type (functional, measurable attribute, non-functional) and level (unit, integration, or system) of all testable requirements detailed by the ILP coursework specification. Each requirement was assigned the lowest effective level of testing to improve the test coverage of the plan. The use of IDs throughout the documents allows for easier cross examination across and within learning outcomes and directly links to the **Test Cases** document where evidence from the tests can be observed. This also prevents needless repetition in the portfolio while maintaining clear accountability for coverage.

Prioritisation of Requirements

As the sole developer working on this project with an estimated 100 hours available, it was necessary to prioritise requirements and allocate testing effort based on risk and impact to achieve effective coverage within the available time. The test plan was created as though development and testing responsibilities were logically separated, even though the project was carried out by a single developer. This allowed each requirement to be linked to a suitable testing level and lifecycle phase.

A software development lifecycle influenced by the V-model (SDLC V-Model - Software Engineering, 2025) is the most suitable overall lifecycle for this project. Requirements can be analysed early and directly mapped to corresponding test activities at the unit, integration, and system levels thanks to the fixed and well-defined ILP coursework specification. Development and testing were in actuality carried out incrementally, with functionality validated as it was implemented, but the V-model provides a suitable framework for ensuring traceability between requirements, implementation, and verification.

There are 3 main requirements which will be the focus of my test plan, below I will go through each stating the risk, importance and allocation of resources needed.

R1 (Very High Risk): Order Validation

The system must be able to classify orders as valid or invalid based off its attributes.

- Impact of failure: Very High
    - Any error in order validation will result in the order being marked as invalid and not being processed.
    - No further downstream functionality is assessed so there will be no flight path calculated for the order.
- Likelihood of failure: High
    - Order validation requires cross validation across many classes and the different nature of the attributes involved in an order result in many ways in which failure can occur.
- Complexity of requirement: High
    - The correctness of classifying invalid vs valid orders is crucial however correctly classifying the reason for which an invalid order failed is not crucial to the running of the system.

    - The complexity remains high as validation logic involves many areas for error such as condition ordering, boundary cases, date parsing, and arithmetic constraints all of which can result in subtle, hard to find errors.

R2 (High Risk): Flight Path Generation
The system must be able to find the shortest path between the given restaurant and Appleton Tower while avoiding no-fly zones.

- Impact of failure: High
    - Failure to generate any flight path would result in a total failure to complete the order given and would have a very high negative impact.
    - However, there are other less impactful ways that flight path generation could go wrong. If the path returned isn't the absolute shortest path possible due to the use of greedy algorithm the impact of which would likely be small, a delivery delayed a few minutes.
    - Failure to avoid no-fly zones could have a severe impact as this would mean the drone is travelling through areas of high student traffic which could raise safety concerns.
    - Violating no-fly zones, central area constraints, or hover requirements would also break specification compliance and would undermine system credibility.
- Likelihood of failure: Medium
    - Many of the components flight path generation relies upon are individually tested at a unit test level, reducing the likelihood of faults should these tests pass.
- Complexity of requirement: High
    - The complexity of this requirement lies more with the interaction of different areas of the software rather than with ensuring the correctness of the algorithm designed to generate the flight path as this is much easier to verify by inspecting the map once the path is overlayed on it.
    - Boundary cases such as a restaurant being close in or enclosed by a no-fly zone would realistically add another level complexity to this requirement however in this coursework we were told to assume this wouldn't be the case, so it won't be covered in this test plan.

R3 (Medium Risk): Drone Movement
The drone must move only in the 16 compass directions and with a set step size and proximity threshold.

- Impact of failure: High
    - Errors in distance calculations or proximity checks can cause infinite loops, unreachable targets, or incorrect path legality. These failures would have a trickle-down effect which would affect navigation and system-level behaviour.
- Likelihood of failure: Medium Low
    - Drone movement logic is mainly mathematically defined and there are a small number of restrictions. The main areas of caution are numerical precision and accuracy in addition to threshold comparisons, both of which can be error-prone if not tested rigorously.

- Complexity of requirement: Medium
    - The complexity lies in thoroughly testing boundary cases of which there are many (such as ensuring the proximity threshold calculation can safely compare points and boundary lines).
    - Behaviour is sensitive to small numerical inaccuracies.

Due to the nature of the ILP as a student project and the intention of it being a simulation of a drone and not having any real-world impact, there are very limited resources. Therefore, here are some of the requirements which would be crucial in a real-world application of this project which will not be covered in this test plan:

- Security:
    - User data (name, card details etc) must be kept private and secure.
- Safety:
    - The drone should fly over buildings and prioritise flying over rooftops rather than streets.
    - The drone should only fly in weather within its capabilities.
- Environmental:
    - The drone should not pose a risk to the bird population.
- Accountability:
    - The restaurant should be paid what they earned from the orders deducting any service charge for use of the drone/system.

Lifecycles

R1: V-model and SRET (Software Reliability Engineer Testing) principles

- Order validation requirements are stable, well defined, and integral to system correctness. A V-model lifecycle is particularly suitable because the validation criteria of the orders can be directly mapped from the specification to integration-level tests.
- A V-model has low flexibility and doesn't require user feedback till the end of the development process. This model is most suited for critical safety systems.
- In this test plan, order validation will be treated as a critical service and such SRET principles would be applied with automated regression testing and no tolerance for failures in order validation. This will ensure reliable and deterministic behaviour of the system.

R2: Incremental Process Model (Incremental Process Model, 2025) with DevOps

- The system will be built step by step and require continuous testing through the development. Navigational legality depends on complex interactions between multiple components making the incremental model the most appropriate lifecycle.
- DevOps is very flexible, very fast and has a low cost of change. It is most suitable applied to a service which may need continuous updates. A requirement of the system is that flight path generation must take less than 60 seconds so having a fast lifecycle is important. DevOps would be used to automate system-level testing and to continuously validate the routing behaviour.

R3: Extreme Programming

- Geometric requirements are small, deterministic, and mathematically defined, making them well suited to an Extreme Programming lifecycle. Unit tests would be written before implementation to specify expected behaviour, especially around numerical thresholds such as the step size and proximity threshold.
- A collection of strong unit tests would ensure early detection of faults and preventing propagation of numerical errors into higher-level navigation logic.

Scaffolding and instrumentation

R1:

- The primary scaffolding change was the extraction of order validation logic from the REST controller into a dedicated OrderValidator service. This separation allowed for validation behaviour to be tested independently of the Spring MVC layer.
- External dependencies such as restaurant data and system time were abstracted behind interfaces and injected into the validator. This enabled deterministic testing of price checks, restaurant availability, and date-based rules without reliance on static data or the runtime environment.
- Instrumentation was applied through the structured OrderValidationResult, which exposes both the overall validation status and a specific validation code. This allows tests to assert that each failure condition is triggered in isolation and that the correct error code is returned, matching the ILP specification.

R2:

- Routing logic was separated from the REST layer and implemented as functions operating on geometric and Region data. This allowed individual components such as heuristic distance selection and no-fly-zone checks to be tested independently from HTTP handling and order validation.
- Region data was given rather than hard-coded, enabling tests to reproduce edge cases deterministically.
- Instrumentation was introduced by writing the generated flight path as a sequence of coordinates and as a GeoJSON representation. This enabled both automated assertions and manual visual inspection of spatial correctness using external tools.
- The system also reports failure explicitly when no valid path can be found, allowing negative scenarios to be tested reliably.

R3:

- Scaffolding was applied by implementing drone movement as geometric operations over simple coordinate objects, independent of routing logic, REST handling, or external data. Methods responsible for calculating next positions, Euclidean distance, and proximity thresholds were kept pure and free of outside influence. Movement parameters such as step size and angular increments were defined as constants (as determined by the ILP specification), allowing tests to validate boundary conditions and numerical accuracy in isolation.
- Instrumentation was achieved by exposing movement outcomes directly through return values. Distance calculations and proximity checks return explicit numerical or boolean results, enabling precise assertions around threshold behaviour.

Risks

R1:

- **Scheduling risk:** Validation logic depends on correct ordering of checks and consistent error handling. Delays in finalising validation rules may require rework of integration tests due to tight coupling between test cases and specification-defined error codes.
- **Development risk:** Because validation spans multiple domain objects (Order, Pizza, Restaurant, pricing rules, dates), insufficient integration coverage could allow conflicting rules to interact incorrectly. Mitigated by designing tests that trigger exactly one validation failure per request, mirroring ILP auto-marker behaviour.
- **Personnel risk**: Validation logic is specification-driven and non-trivial; misunderstanding of edge cases (e.g. total price calculation including delivery fee) could lead to systematic test or implementation errors. Risk is reduced through explicit reference IDs and traceability between requirements and test cases.

R2:

- **Development risk**: Automated tests are limited in their ability to fully verify spatial legality of paths (no-fly zones, central area constraints). Reliance on partial manual verification (GeoJSON visual inspection) introduces subjectivity and reduces repeatability. Risk accepted due to coursework scope and mitigated by focused integration tests checking key constraints.

R3:

- **Development risk**: Errors in numerical thresholds or step-size calculations may only manifest indirectly during path generation. Risk mitigated by exhaustive unit testing of boundary values and reuse of movement logic in integration tests.

Evaluation

Overall, the greatest risk lies in Order Validation and Flight Path Generation, as both involve coordination between multiple components and require adherence to strict specification rules. Drone Movement presents minimal risk due to its simplicity and suitability for isolated unit testing. The proposed bottom-up testing strategy ensures that foundational behaviour is verified early, reducing the likelihood that defects propagate into higher-level system behaviour.

Bibliography

Incremental Process Model. (n.d.). Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/software-engineering/software-engineering-incremental-process-model/

SDLC V-Model - Software Engineering. (2025). Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/software-engineering/software-engineering-sdlc-v-model/