Test Planning

Since the project code has already been implemented, I won't be using a test driven development approach and instead will be using the Waterfall model. In the Waterfall approach, development progresses sequentially through phases, with testing occurring after implementation. This aligns well with the current state of my project, where the focus is on thoroughly validating the implemented functionality.

Test Strategy

The strategy focuses on verifying important parts of the system:

- R1: The system should be able to validate orders accurately
- R2: The system should find the most optimal path without entering no-fly zones
- R3: The system should adhere to the 60 second runtime constraint

There are other parts of the system that are not within my scope of testing for this portfolio such as security and privacy of the users and their data.

Priority and Pre-requisites

R1:

- Priority - high
- This is a functional system level requirement that is important for maintaining system integrity.
- Order validation is a basic and core system requirement to make sure that invalid orders are noticed and excluded from processing. This directly impacts subsequent operations like the pathfinding algorithm and file generation, hence this is important to address first.

To test this requirement, I will be using:

Unit testing

- Validate individual components involved in order validation to make sure each part operates as expected in isolation and catch any errors early on in the system lifecycle. This aligns with the **principle of sensitivity** from Chapter 3, which states that it's better for the system to fail consistently rather than unpredictably. Detecting faults early helps prevent them from being found later in the process, where fixing them would take more time and effort.
- Test edge cases, for example:
    - minimum and maximum pizza counts (1–4 pizzas)
    - Digits in credit card information

System level testing

- Verify that invalid orders are correctly flagged and excluded from downstream processes, and only valid orders proceed to pathfinding and file generation
- Simulate various scenarios (such as invalid orders due to missing data or incorrect format) to make sure the system handles errors gracefully and assigns appropriate validation codes

By combining detailed unit testing with comprehensive system-level testing, this approach makes sure that all aspects of order validation are fully tested, both individually and as part of the overall system to minimise the risk of undetected issues in production.

Boundary Value Analysis

Validation fields like pizza count, CVV, and card number involve distinct input ranges where testing just inside and outside the boundary is important for robust validation logic

Example:

- CVV: Test with 2 digits (invalid), 3 digits (valid), and 4 digits (invalid)
- Pizza count: Test with 0 pizzas (invalid), 1 pizza (valid), and 5 pizzas (invalid)

Integration testing

This step is critical to confirm that data flows correctly between components and that the system flags invalid orders appropriately.

The focus of integration testing for R1 includes:

- Data Flow Validation
    - Ensure the credit card, pizza, and restaurant validation components correctly share and process data for a given order
- End-to-End Order Validation
    - Test scenarios where multiple validation failures occur simultaneously

Pre-requisites:

Credit Card Information

- CVV must be exactly 3 numeric digits
- Card number must be exactly 16 numeric digits
- Expiry date must follow the MM/YY format and not be in the past

Pizza information

- An order must include at least one pizza
- Each pizza must have a name and a price greater than zero
- The total price of pizzas plus the order charge must match the priceTotalInPence field
- The number of pizzas in an order must not exceed 4

Restaurant Information

- Pizzas in the order must exist in the menu of a defined restaurant
- All pizzas in an order must come from the same restaurant
- The restaurant must be open on the day of the order
- Prices of pizzas in the order must match those listed in the restaurant’s menu

R2:

- Priority - high
- The pathfinding algorithm is important for generating delivery routes that avoid no-fly zones and minimise travel time. As outlined in Chapter 3, the main operational efficiency of the system depends on accurate and legal path generation.
- This is a functional system-level requirement that is important for efficient and compliant delivery operations. It guarantees the drone avoids restricted areas while calculating the shortest valid path to the delivery location. Since the orders are fetched from a REST API, I will not simulate an exhaustive number of orders but will focus on various no-fly zone scenarios to comprehensively test this requirement.

To test this requirement, I will be using:

Simulation testing

Simulate scenarios to test edge cases and unusual configurations of no-fly zones

- Standard Scenario
  - Test with a realistic number of no-fly zones, as provided in the example, to know the drone navigates around them correctly
- No No-Fly Zones
  - Test that the system generates a direct and optimal path without any obstacles
- High-Density No-Fly Zones
  - Test with a large number of no-fly zones
- No-Fly Zones Over the Delivery Location
  - Test situations where no valid path exists because no-fly zones cover the delivery point. Verify that the system handles this gracefully
- Weirdly Shaped No-Fly Zones
  - Test with irregularly shaped no-fly zones
- Intersecting No-Fly Zones

Unit Testing

- Validate LngLatHandler to make sure it
  - Accurately identifies points within or outside no-fly zones
  - Properly calculates distances and movement constraints around no-fly zones.

Boundary Testing

- Test paths that graze the exact boundary of a no-fly zone
- **Fuzz Testing:** Feed the system random, unexpected data to have robust error handling and stability.

Pre-requisites:

- AStarFlightPathSolver
  - Pathfinding logic must correctly incorporate no-fly zone boundaries when calculating paths
- LngLatHandler
  - Accurate calculations for detecting whether a point is inside or outside a no-fly zone
- Input Data
  - No-fly zone configurations for testing each scenario
  - Delivery locations and starting points that allow for varied pathfinding challenges
- GeoJSON Output
  - Output coordinates to visualise the path to inspect for correctness of constraints

R3: Performance requirement

- Priority - medium
- This is a system-level non-functional requirement focused on making sure the system sticks with the 60-second runtime constraint for generating paths and processing orders. While meeting this constraint is important for scalability, responsiveness, and user satisfaction, the priority is medium because finding a path, even if it takes slightly longer, is more important than failing to find one entirely. The principles from **Chapter 3** support this perspective:
  - **Sensitivity:** Makes sure predictable outcomes by prioritising correct pathfinding results over occasional adherence to the runtime constraint. It is better to consistently find valid paths than to fail unpredictably due to strict time limits.
  - **Restriction:** Simplifies the focus by addressing the core problem of finding valid paths first, while optimising the runtime can be iterative and an incremental improvement

To test this requirement, I will be using:

Performance testing

- Measure the performance of the pathfinding algorithms under:
  - Standard conditions with typical number of no-fly zones
  - High-stress scenarios, such as large number and complex no fly zones
- Record and analyse cases where the runtime exceeds 60 seconds, ensuring valid paths are still generated

Simulation testing

- Test the system under different geographic complexities
  - Simple and complex no-fly zone configurations
  - Orders with different geographic distributions
- Evaluate the system's ability to balance pathfinding correctness and runtime efficiency

Integration testing

Test the interaction between the pathfinding algorithms and the validation processes, to make sure the system:

- Properly incorporates no-fly zone constraints into the generated paths
- Maintains performance across varying inputs during end-to-end workflows

Verify that the order validation, pathfinding, and output generation processes are seamlessly integrated and do not introduce unnecessary delays

Pre-requisites:

- Optimised Algorithms:
  - Ensure pathfinding and validation algorithms are optimised for performance
- Logging and Monitoring:
  - Implement detailed logging to track runtime for different components

Strengths and Weaknesses

The test plan provides a strategy for the system to meet its core requirements (R1, R2, R3). However, some potential omissions and limitations exist. For R1, while random inputs cannot exhaustively cover all edge cases, I will prioritise critical scenarios such as invalid credit card details for robust validation testing. For R2, testing will focus on key no-fly zone scenarios like large amount of no-fly zones and irregular configurations. Its not possible to simulate every layout, but this approach targets meaningful and realistic edge cases. For R3, performance testing will simulate varying conditions effectively, but some highly specific runtime challenges might remain unexplored. Despite these limitations, the chosen testing plan is good for verifying the system’s functionality and performance under practical scenarios.

Scaffolding and Instrumentation

R1

Scaffolding will be needed to generate random inputs for validation, such as

- Randomly generated credit card details (valid and invalid)
- Pizza orders with varying counts and prices

Instrumentation will be required to check if outputs such as validation codes meet the expected specifications

- Logging validation results to have accuracy
- Analysing outputs to identify patterns of failure for refinement

R2

Synthetic no-fly zone data and delivery locations will need to be designed for testing

- Standard configurations
- Edge cases, such as no-fly zones completely blocking a delivery
- Randomly generated complex scenarios to test robustness

Instrumentation will include a tool to

- Visualise paths to confirm they comply with constraints
    - We can use geojson.io for this
- Log details of the pathfinding algorithm's decisions for analysis
    - We can use the json output files for this

R3

Scaffolding to simulate varying input sizes and complexities

- Create test cases with varying numbers of no-fly zones
- Generate inputs that push runtime to the edge of the 60-second limit

Instrumentation:

- Logging and monitoring runtime metrics, such as start and end times for pathfinding and validation processes
    - System.nanoTime()
- Analyse bottlenecks in subsystems using runtime breakdowns

The scaffolding and instrumentation for R1, R2, and R3 provide a solid foundation, but there are still some limitations. For R1, it won't be possible to generate random inputs for every edge case, but focusing on critical scenarios like invalid credit card details and boundary pizza counts should provide robust validation. For R2, while not all possible no-fly zone configurations can be tested, standard and key edge cases will sufficiently evaluate pathfinding. For R3, varying input sizes will simulate runtime constraints effectively, though a few highly specific scenarios might be untested. Overall, this approach should reliably cover the most impactful scenarios within practical limits.

Process and Risk

R1

- **Process:** The scaffolding for generating random inputs should be implemented early in the lifecycle. This is so that individual components can be tested thoroughly before integration. Instrumentation for logging and analysing validation results should follow once scaffolding is in place.
- **Risk:**
    - If the generated inputs fail to cover key edge cases, some validation issues may go undetected, whihc can cause expense and delay in integration testing

R2

- **Process:** Creating synthetic no-fly zone data should be done early and in parallel with other activities. Visualization tools, such as geojson, will be used to ensure pathfinding results align with expectations.
- **Risk:**
    - If synthetic data is unrepresentative of real-world scenarios, the pathfinding algorithm may perform poorly in operation.
    - Visualisation outputs may be insufficient for identifying subtle issues in pathfinding

R3

- **Process:** Scaffolding for performance testing, including workload generation, should be built once pathfinding and validation subsystems are functional. Runtime monitoring and analysis tools can be introduced concurrently with system-level testing and refined iteratively during performance tests.
- **Risk:**
    - Workload simulation may fail to fully replicate high-stress, real-world conditions
    - Runtime metrics might not capture all bottlenecks which can lead to undetected inefficiencies