Requirement levels and testing approaches

Unit-Level Requirements

OrderValidator Class:

- Validate the general structure and attributes of an order:
    - Correct credit card information
        - Make sure the CVV is exactly three digits and contains only numbers.
        - Confirm the credit card number is 16 digits long and contains only numbers.
        - Check that the credit card expiry date follows the "MM/YY" format.
        - Verify that the expiry date is not in the past.
    - Correct pizza information
    - Correct restaurant information
    - The associated restaurant is valid and operational.
- Apply appropriate validation codes for any invalid order attributes.
- **Pathfinding Class:**
    - Implement the A* algorithm to calculate the most optimal route.
    - Stick to the 16 compass directions during pathfinding.
        - This is stated in the Direction enum
    - Validate that all paths avoid no-fly zones.
- **LngLatHandler:**
    - Get the Euclidean distance between two positions.
    - Test the isCloseTo method to ensure it correctly check for proximity (< than SystemConstants.DRONE_IS_CLOSE_DISTANCE).
    - Verify the isInRegion method for detecting whether a point is inside a given polygon, including edge cases.
    - Test that the nextPosition method calculates the correct next position based on angle and movement distance.
- **FlightPath Class:**
    - Generate a complete flight path for each order, including forward moves to the delivery location and return moves to the starting point.
    - Translate a list of coordinates into individual FlightMove objects for the drone's movement.

    - Add hover moves at both the delivery goal point and the starting point to signify delivery actions and completion.
    - Reverse the path after reaching the delivery location to construct the return journey.
- **FlightMove Class:**
    - Represent a single movement of the drone, including the starting and ending coordinates.
    - Indicate whether the movement is part of the forward path to the goal or the return journey.
    - Capture metadata about the associated order for each movement step.
- **Output JSON and geoJSON files:**
    - The lightpath that the drone takes on the sepcified day in JSON and geoJSON formats.
    - Make sure output files (deliveries, flightpath and geojson) are in the specified formats.
    - Test for proper serialisation of order and path data into JSON format.

Integration-Level Requirements

- **REST API Integration:**
    - Have proper interaction between GetClient and REST endpoints for:
        - Fetching restaurant details.
        - Retrieving no-fly zone data.
        - Fetching daily orders.
    - Handle errors such as timeouts, invalid endpoints, or incomplete data gracefully.
- **Order Validation Workflow:**
    - Integration between Orders, OrderValidator to:
        - Validate orders against all constraints.
        - Flag invalid orders with appropriate order validation codes.
        - Exclude flagged orders from the delivery list.
- **Pathfinding and Execution:**
    - Integration of AStarFlightPathSolver with:
        - LngLatHandler for geographic calculations and movement validations.
        - FlightMove and Hover for generating valid move sequences.
    - Ensure flightpaths respect no-fly zones and central area boundaries.
- **Geographic and Drone Behavior:**
    - Integration of LngLatHandler with FlightPath to:
        - Calculate distances between drone positions.

  - Validate positional accuracy when the drone enters, hovers, or leaves specific areas.
  - Ensure polygon detection works accurately with no-fly zones and regions.

Test Approaches for chosen attributes

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.

I will use several approaches to make the system reliable and have good performance.

- **Unit Testing:** Validate critical components like LngLatHandler for accurate distance calculations and region checks, and CreditCardInfomationCheck for ensuring proper order validation. Test each method in isolation to guarantee correctness before integration.
- **Integration Testing:** Verify the interaction between subsystems, such as FlightPath integrating with AStarFlightPathSolver to generate optimal routes while adhering to constraints like no-fly zones and move limits.
- **System level testing:** Validates the functionality of the entire system, including order validation, pathfinding, and output file generation, under realistic conditions. This ensures the system meets all functional requirements, including error handling and compliance with constraints.
- **Boundary testing** is particularly valuable for validating edge cases such as coordinate boundaries for no-fly zones, amount of no fly zones (if 0 or to many), maximum and minimum pizza counts, or amount of orders.
  - **Fuzz Testing:** Feed the system random, unexpected paths and data to ensure robust error handling and stability.
- **Simulation Testing:** Evaluate the system by testing:
  - Feasible paths: Check the drone reaches its destination within constraints.
  - No feasible paths: Validate the system appropriately handles scenarios where the drone cannot complete the delivery.
  - Extreme no-fly zones: Simulate scenarios where the drone cannot reach the destination due to extensive no-fly zones.
- **Performance testing:** The system should also be tested to ensure the 60-second runtime constraint is consistently met.
- **Black-Box Testing:** This technique evaluates the system based on its external behaviour without considering internal code structures. For example, validating the end-to-end functionality of order validation and pathfinding by using synthetic data to simulate real-world scenarios.

Together, these approaches allow for thorough evaluation across functional, integration, and user-facing components.

Appropriateness of chosen testing approach

The chosen testing approaches provide a comprehensive strategy to make sure the system’s reliability, performance, and usability, though each has its strengths and limitations.

The process begins with unit testing to validate individual components, followed by integration testing to confirm smooth communication between subsystems, such as the flight path and pathfinding logic. System-level testing is conducted afterward to evaluate the functionality of the entire system under realistic conditions.

Unit testing verifies components like distance calculations and credit card validation but does not address interactions between classes. Boundary testing is effective for edge cases, such as maximum pizza counts or excessive no-fly zones, but designing extensive test scenarios can be time-consuming. Fuzz testing adds robustness by introducing random inputs, though it may generate unrealistic scenarios, leading to false negatives. Simulation testing evaluates realistic delivery scenarios, such as feasible and infeasible paths or too many no-fly zones, but it may not capture realistic scenarios that are not testable. Integration testing ensures smooth communication between subsystems, such as the flight path and pathfinding logic, but it may overlook low-level issues in isolated components. Performance testing confirms the system meets the 60-second runtime constraint, but realistic load testing can be challenging without proper infrastructure. Despite these limitations, this testing approach covers most aspects of the system.