**Requirements Document**

**1. Functional Requirements**

**1.1 Order Processing**

* The system must retrieve all orders for a specified date.
* Orders must be validated against restaurant menus, opening hours, and payment details.
* Each order must be flagged as either:
  + DELIVERED if successfully delivered.
  + ValidButNotDelivered if valid but not delivered due to it breaking constraints (e.g., exceeding maximum drone moves).
  + The correct OrderStatus if the order is invalid.

**1.2 Flight Path Calculation**

* The drone must start and end at Appleton Tower
* The flight path must that is generated must:
  + Deliver pizzas to valid orders in order.
  + Avoid no-fly zones.
  + Only leave or enter the central area once in each path (from Appleton to restaurant for example)

**1.3 Output File Generation**

* The system must generate three output files:
  + deliveries-YYYY-MM-DD.json: Details for each order, including status and cost.
  + flightpath-YYYY-MM-DD.json: A record of all drone moves, including its coordinates and angle of direction.
  + drone-YYYY-MM-DD.geojson: A GeoJSON representation of the drone's flight path for visualization.

**1.4 Error Handling**

* Invalid input parameters, such as incorrectly formatted dates or inaccessible URLs, must result in an appropriate error message and program termination.

**2. Measurable Quality Attributes**

**2.1 Performance**

* The system must complete its operations (order processing, flight path calculation, and file generation) within 60 seconds on standard hardware.

**2.2 Accuracy**

* The drone must strictly avoid no-fly zones and deliver orders to restaurant-provided coordinates with a precision of at least four decimal places.
* The GeoJSON file must approximate calculated flight path coordinates.

**2.3 Reliability**

* All valid orders for the specified date must be either delivered or flagged appropriately.
* The system must handle edge cases, such as orders that exceed the maximum allowable moves or orders to restaurants that don’t exist.

**3. Qualitative Requirements**

**3.1 Compliance with File Naming and Format Standards**

* File names must:
  + Use only lowercase letters.
  + Use hyphens (-) instead of underscores or spaces.
  + Follow the ISO 8601 date format (YYYY-MM-DD).
* JSON files must follow standard rules for json format of the data.

**3.2 Extensibility**

* The system must allow future enhancements, such as adding new no-fly zones, modifying central area boundaries, or updating restaurant menus, with minimal changes to the codebase.

**4. Levels of Requirements**

**4.1 System-Level Requirements**

* End-to-end functionality:
  + Process orders for a given date.
  + Calculate and execute a flight path.
  + Generate output files in the correct format.
* Performance: Ensure runtime is under 60 seconds.

**4.2 Integration-Level Requirements**

* Interaction between the following components:
  + OrderValidator: Validates orders using restaurant data.
  + FlightAStar: Calculates optimal flight paths while avoiding no-fly zones.
  + PathsMapped: Caches previously calculated paths for efficiency.
  + FileHandler: Handles file generation.

**4.3 Unit-Level Requirements**

* Each core method must function independently:
  + getValidOrders: Filters and validates orders.
  + getPathForDay: Calculates a full day’s flight path.
  + getMovesList: Generates a sequence of moves from a flight path.
  + recordDelivery, recordMove, recordGeoJson: Generate output files correctly.

**5. Testing Approach**

**Validation Testing -** Validate that orders are correctly filtered by date, payment details, and restaurant availability.

**Boundary Testing**

* Test flight path calculations near the edges of the central area and no-fly zones.
* Verify the system’s behaviour with exactly 2000 moves.

**Performance Testing -** Test runtime using various order volumes and restaurant configurations to ensure calculations are completed within the 60-second limit.

**Compliance Testing -** Validate output files for correct naming conventions, attribute naming, and data formats.

**Edge Case Testing -** Test scenarios with no orders, invalid orders, or orders with unreachable destinations.

**Integration Testing** – Once unit tests are completed integration between these different classes will need to be tested to ensure that they interact as intended.