Requirements to the Pizza Dronz software:

Functional Requirements

1. (System Level) Flight control: The software must be able to control the drone's movements.
   1. (Integration level) The drone should not cross the Central Area boundary multiple times thus violating the cannot leave central area after entering back in rule.
2. (System Level) Connection: The software must be connected securely to the REST Server
   1. (Integration level): The software must retrieve order data and restaurant/menu data from the REST Server
      1. (User level): Input validation. The software should be able to validate orders and handle non valid orders without disrupting the program
      2. (User level): The software should be able to calculate the final cost of the delivery
3. (System Level) Navigation: The drone must be able to navigate and follow a predetermined flight path.
   1. (Integration level): The flight path needs to be sensible
   2. (Integration level): The software must not cross populated areas or “no fly zones”
   3. (Integration): The drone must have a capability to return to Appleton tower, when done with the order.
      1. (Unit level): The program should be robust and reliable, and handle errors without disrupting the main loop, which if not handled, could lead to the drone crashing
4. (System Level) Battery management: The software must be able to monitor and manage the drone's battery life to ensure it does not run out of power during flight.
   1. (Integration level): The software must be able to calculate the battery needed for the flight and check if it can finish the trip without running out
5. (System Level) Data export: The software must be able to export data, such as the flight path or the delivery log.
   1. (Integration level): Data integration: The software must be able to integrate and process data from multiple sources, such as sensor data, GPS data, and remote control inputs.
6. (System Level) Autonomy: The software must have the capability of flying the drone autonomously according to pre-defined instructions and rules.

Measurable quality attributes

1. The time it takes to generate flight path should be measured as a quality attribute and be less than the assignment’s standards

* 1. Less than 60 seconds

1. The runtime for order processing should be very short
   1. Less than 1 second, Very short

Qualitative requirements

1. If the flight path seems sensible

2. The program outputting three files, deliveries.json, flightpath.json, drone.geojson

* 1. Deliveries.json should have the following:
     1. *orderNo* - the eight-character hexadecimal string assigned to this order in the orders REST service endpoint
     2. *outcome* - the OrderOutcome value for this order, as a string
     3. *costInPence* - the total cost of the order as an integer, including the standard £1 delivery charge
  2. Flightpath.json should have:
     1. *orderNo* — the eight-character order number for the pizza order which the drone is currently collecting or delivering
     2. *fromLongitude* — the longitude of the drone at the start of this move
     3. *fromLatitude* — the latitude of the drone at the start of this move
     4. angle — the angle of travel of the drone in this move18
     5. *toLongitude* — the longitude of the drone at the end of this move
     6. *toLatitude* — the latitude of the drone at the end of this move and
     7. *ticksSinceStartOfCalculation* — the elapsed ticks since the computation started for the day - every record will have a higher value than the previous one and records the duration this move calculation took
  3. Drone.geojson should have:
     1. A geojson format which contains one Feature: LineString

Testing:

Unit Testing

The path finding function will be tested through unit testing by feeding the route finding function coordinates of Appleton tower and that of a restaurant. I would use geojson.io to check if my drone pathing algorithm is working properly.

Some tests I could do are:

- Check if it successfully goes from Appleton tower to Domino’s pizza

- Create a unit test when the starting point and the end point are very close together

Also by using the REST server’s <https://ilp-rest.azurewebsites.net/orders>, I would be able to create a unit test to process orders and check if all the functions are working properly.

- Check if each error is outputted correctly (eg. Does not output pizza combination error when credit card number error)

- Check if the errors don’t break the code

- Create a unit test so that the program correctly calculates the delivery cost for each order

Integration Testing

- Check if the software can process one day worth of orders and validate them properly

- Check if the drone does not pass the central area boundary multiple times during a trip

- Create tests to see what the drone does when it doesn’t have enough battery to complete a trip

- Create a test to see if the flight path it creates is correct and sensible

- Check if the program is able to output the necessary json and geojson files

Performance Testing

- Test if the software can process any day worth of data and create flight paths under 60 seconds in total

- Test 10+ days of orders to check if it completes the task under 1 minute for each day.

Not appropriate testing methods

Some testing methods that won’t be appropriate for this specific use case is:

Usability testing – The software does not need a UI, as a result the usability of the program does not need to be tested

Stress testing – The program is not meant to process many orders at once, so there is no need to test unexpected loads of orders to the program

Security testing – There will be no security measures in the software, as this is just a theoretical software that delivers code. So a security test will be meaning less

Additional tests outside the area of Edinburgh and the restaurants: Anything outside the restaurant area will be meaningless as this is a software that is only made to be used inside the boundary covering George Square.