Level of Requirements

Here we will consider **system**, **unit** and **integration** requirements that were considered for the ILP project. This will highlight a sample of all the requirements and which level of requirement they may fall under. The project had split into **two** separate submissions in total, where the first submission considered:

* Creating the LngLat record class for allowing object instance methods
* Retrieval of the restaurant/menu data and the central area data from the rest server.
* A order class that validates the Strings of pizza order items with the restaurants, creating malfunction exception messages for any wrong pizza orders.

The second submission considered all of the above and also some of the requirements below:

* No fly zones
* More order validation
* Drone flightpath planning
* Calculating delivery outcomes for each order
* Performance aim of running and outputting 3 files within 60 seconds

First submission

For the first submission, there are some different unit, system and integration levels of requirements being met. Note that there will be less of integration or system and more unit due to the simple complexity of the system so far. Here are some examples of different levels for their requirements:

Unit:

* The **LngLat** record class **methods** were **checked individually for functionality**, either by **unit** tests or **manually** using the System.Out.Println() in built java method – most methods contained in this class were simple functions that were easily tested for as they would usually return a Boolean value, double value or a new LngLat object as per the project requirements. We expect the method to return something and could easily check if that value is correct or not. For example, checking the double value of the distance to another LngLat point.
* The **Restaurant class method** for taking in an URL that points to the rest server and could retrieve the relevant data could be checked for **functionality** – the level of testing here is unit as no other methods are required for the first submission, it could check for if the returned values from the rest server length is not null.
* The **Order class method** for validating the order item Strings can be tested at an **unit** level – Order items can be generated for Junit tests to test for if the order method can catch the wrong orders. This can also be at an integration level later in course submission 2 where we have access to all the orders.

Integration:

* The testing of the LngLat method inCentralArea() requires the use of the method from the CentralArea class, therefore making it a **integration** level testing – to test this method, we need the getCentralArea() method to be working as well, **built on top** of the unit testing already done. The getCentralArea() method will need to be working correctly to receive the rest server data and convert it into coordinate information for LngLat. Manual checking from printing results can verify correctness.

System:

* Verifying if order items were valid is **system** level at this stage, as everything was still quite simple. This required **most of the classes and their methods** for submission 2 to be functional. – Requiring the retrieval of the rest data for the restaurant, their menus, order validation on the order item strings and printing out the correct malfunction error message if a fault was detected, this was done **manually**.

Second Submission:

A lot more integration and more complete system tests may be carried out as we have more classes and methods working together for the final system. Unit level testing is building upon the submission two testing already in place.

Unit:

* Checking a valid input date requires methods, which need to be **unit** tested for correctness – there are **junit tests** in place to assert that the expected values will match the actual values returned. This is unit level as only one method at a time is being tested.

Integration:

* Determining the delivery outcome for all orders is integration level requirement – we had to **integrate the use** of retrieving the orders, restaurant and their menus from the rest server, all methods from the delivery class had to be used after unit testing. This gave us a full list of valid and invalid orders.
* Calculation of the final flightpath is integration level requirement – we had to use the methods from the delivery order validation, all of the CalculateFlightpath class methods, use of restaurant and order data retrieval, checking the input date and the LngLat methods for movement and drone coordination. This was **tested manually** and also using **junit**. This is close to system level, but since it doesn’t require the full system to be working yet to test, it is still integration level.

System:

* To verify the **performance** of the system to verify the functional requirement of running the program **under 60 seconds**, the system was ran successfully several times on all orders provided to ensure that it achieved the performance required. The **average time** was taken after 4 runs and it was safely under the minimum requirement.
* The drone flightpath could be checked for any faults with the flightpath by using the **geojson output files** for order of any given day. This is system level due to the use of **all the program class methods** to create the resulting output files from the main App class. The geojson data can be viewed with the all.geojson data provided for the school campus to see if any drone flight rules were broken when flightpath was plotted out on the website. This can be checked manually.