Test Plan

This document contains the testing plan for the ILP Pizza Dronz project. Three requirements will be selected to demonstrate the understanding of this learning objective. As only three will be covered, one of each level requirement will be selected; unit, integration and system. The requirements covered will be:

1. The drone should be considered to be close to a location if it is within 0.00015 degrees to it.

2. The orders must be validated correctly. *(The system must ensure that the orders coming in are valid and if they are not, they will not be carried out. The system will go through each order when it is placed and check if all aspects of the order are valid such as an acceptable order date and of the restaurant is available on the day of the order. If the order is invalid, it will indicate this with an error message and this order will not be placed.)*

3. The system must not allow for the drone's flight path to enter any no fly zones.

Priority and prerequisites

1. **The drone should be considered to be close to a location if it is within 0.00015 degrees to it.**

This is a Unit level functional requirement which should be of medium priority. Whilst it does not directly impact any persons safety or affect large amounts of the code, if incorrect it could lead to slightly incorrect routing which could lead to the drone landing in an incorrect place.

This requirement involves a pythagorean formula for calculating the distance between two points. To verify this calculation, a simple unit test can be constructed to check that two given points are producing the correct output. This can be implemented at the beginning of development as a black box test as the infrastructure of the system does not need to be known. Some synthetic data will need to be constructed for the unit test, such as two coordinates that are known to be close to one another or not.

The requirement will take a set of two coordinates (lng1, lat1), (lng2, lat2) as an input, and will return a boolean value as to whether or not the calculated distanceBetween value is less than to 0.00015 degrees. It should only return true when the distance between the two coordinated is in fact less than 0.00015 degrees and false otherwise.

Throughout the coding process, redundancy checks should be carried out as it will both help find errors sooner and will improve code maintainability for the next developers.

As this is only a small unit level test, the tasks that may need to be scheduled into the plan include:

- Generating the synthetic data for the test.

- Writing the unit test itself.

2. The orders must be validated correctly.

This is an integration level functional requirement that should have high priority. There are multiple potential ways in which an order can be wrong and some could lead to customers making incorrect payments or the drone being asked to carry more pizzas than it can which may lead to a crash. This therefore is a potential safety hazard if not correct, and so is high priority.

As it is high priority, we should be aiming to use as least two different Test and Analysis (A&T) approaches. We shall once again be constantly using automated redundancy checks, for example static type checking. This will keep the code more efficient and more maintainable throughout development. We will also apply the principle of partitioning, to make testing and analysis significantly easier and allow each section of the order to be tested individually.

To partition this requirement, we can test all of the individual order requirements as unit tests. For instance "The CVV given must be 3 digits long" can be black box tested as a Junit test. We also do not have to check every number in the world for this, we will just check edge cases. In this case, we can check a 3 digit CVV, a 2 digit CVV and a 4 digit CVV. This will allow us to cut down on the amount of tests and their computational time. Then integration level testing will be required, to see if the correct error is flagged when all of the smaller tests are working together to validate an order.

The input of this requirement will be an Order class and the output will be a boolean value orderValidation, indicating whether or not the order is valid, and an order orderStatus that will either display why the order is invalid or, if the order is valid, it will display whether or not it has been delivered.

Several tasks will be added to the schedule in order to test this:

- Generation of the synthetic data that will test case boundaries for the individual unit tests.

- The creation of the unit tests.

- Some sort of inspection of the order validation to ensure that the orderValidation and orderStatus codes cannot be invalid pairings (i.e. orderValidation is true but orderStatus says invalidCVV) and that it is not possible for an Order to have two orderStatuses.

- An exhaustive check of every possible type of Order that would produce every combination of orderValidation and orderStatus.

3. The system must not allow for the drone's flight path to enter any no fly zones.

This is a system level functional requirement that should have high priority. Areas can be allocated no fly zones both for privacy concerns and for the safety of the public, for instance if it is a crowded area. Avoiding these is therefore very high priority, as it could concern the safety of the community if a drone were to have a mechanical fault.

Given it is a high priority requirement, we will use at least two distinct A&T approaches. We will perform automated redundancy checks such as static type checking throughout development, to ensure code efficiency and maintainability during the development process. Additionally, we will implement the principle of partitioning to allow more straightforward testing and analysis, and allow each section of the order to be tested individually. We will also then check the flightpaths visually, to easily spot if there has been a large error made in the code.

To partition this requirement, we will check the edge cases and boundary values, to avoid having to check every possible flightpath with every possible no fly zones. This could include seeing if the code will work if there is just one no fly-zones, zero no-zones or multiple. It will also check that the flightpath is not affected if the no-fly zone should not be in the way of the drone and its destination, and what happens when it is.

We will visually test this by getting the GeoJson data produced by the flightpath and the GeoJson data of where the no-fly zones are and put them into the GeoJson website. This will allow us to manually see whether or not the flightpaths are ever crossing into no-fly zones. This will take longer than the other tests but is still important to do as this is a high level requirement, and more time should be dedicated to it.

The input of this requirement will be the starting and ending coordinates for the drone, and the no-fly zone coordinates that will need to be avoided. The output should be a flightpath, in three different Json formats, that do not cross into these no-fly zones at any point.

Several tasks will be added to the schedule in order to test this:

- Generation of the synthetic data that will test case boundaries of the no-fly zones.

- An inspection of the flightpath and no-fly zones to ensure that they do not intersect.

- An exhaustive check of different possible combinations and placements of no-fly zones to see if the generated flightpath still does not intersect.

Process and risk

1. **The drone should be considered to be close to a location if it is within 0.00015 degrees to it.**

As this is a unit level requirement, the synthetic data needed for the testing is not that difficult to create. Additionally, as we only really need to check boundary cases (the distance is 0.00015, its larger than 0.00015 and its smaller than 0.00015) this test will not take much time to create at all. There would be no need to check more values than these three test cases, as it would just increase the time and would be very unlikely to uncover an error. Th creation of this test would take at most an hour.

There is a small risk involved with this test, as it will not take a long time to implement and it is very likely to catch any errors in this piece of code. Even if the test is faulty, as mentioned before, the drone would either be fractionally off of its landing spots or it would be very obviously far from them which would be easily caught when visually checking flight plans later in development.

2. The orders must be validated correctly.

This will require writing several unit tests that use synthetic data and then creating more synthetic data to test all possibilities of outputs when the different checks are integrated together. This process may takes several days to complete. The process also get much longer the more errors that you come up with. The REST server has several orders and error messages already supplied, so this will help save time, but it will still likely be a very time consuming process.

There are many risks to consider with this requirements testing. As stated before, the more errors that could occur with an order the longer the order validation testing process becomes. This means that a large amount of time should be allocated to this issue, in case more errors than expected are found. If invalid orders are not caught by this validator, or are labeled incorrectly then this could lead to a huge backup of orders that are either accidentally being placed or valid ones that have been payed for not being sent. This would be a significant issue with the system, and could lead to huge monetary losses or legal problems. Therefore, this must be tested thoroughly, and every possible order type tested.

3. The system must not allow for the drone's flight path to enter any no fly zones.

This requirement is by far the largest as it requires many other components to be developed to test, and it is a difficult problem even when these components, such as the generation of a flightpath, are complete. This may end up taking a week or weeks to complete and fully test. To test all different types of scenarios, a lot of synthetic simulations will need to be created, which takes up significant time. Using GeoJson data from the flightpaths and no-fly zones to visually check if the specification holds up in different scenarios also takes up much time and resources. As this is a high priority requirement, a very large amount of the schedule should be dedicated to ensuring that this is adequately tested.

This will present many risks, it could easily take up more time than initially thought depending on the amount of scenarios that need to be tested. This could then lead to other requirements on the schedule not being tested adequately. There are also risks that even if completely tested in a simulation, the simulation does not take into factors such as wind speeds or bird strikes which could push it into these zones. There is also a risk that the flightpath generation becomes very complex to calculate and puts a large computational strain on the whole system. The algorithm dedicated to doing this should be optimised as much as possible within the given timeframe.

Scaffolding and instrumentation

1. **The drone should be considered to be close to a location if it is within 0.00015 degrees to it.**

This is just a simple unit test and so does not require any scaffolding or instrumentation.

2. The orders must be validated correctly.

This will require a simulation of the order validation to be tested. This scaffolding will need to be scheduled early in development process so that testing can be carried out sooner and bugs can be resolved. This will also require some synthesised orders that can be tested, as well as potentially some real orders from the REST server data to ensure that this is integrated correctly.

3. The system must not allow for the drone's flight path to enter any no fly zones.

This will require a simulation of the entire routing system. This will need to be complete as soon as possible in the schedule as testing this is likely to be a long process. Synthesising the different scenarios will also take quite some time, and be needed for testing. This includes different no-fly zones and different start and end points. Additionally, actual data from the rest server will need to be tested on, so that the integration is proven to work. The outputs of these simulations will need to be observed and put into GeoJson for manual visual checks.

Evaluation of the scaffolding and instrumentation

By using a combination of synthetic test data and real data from the REST server, the proposed instrumentation proves a good level of testing. The testing strategy should provide adequate coverage, and this code coverage can be tested with different analysis tools to ensure it is sufficient. The testing could be further improved by allowing Beta testing from actual users, to see how the system would actually be used and see if any errors were overlooked in the synthetic data. There can also be a certain level of manual random testing from the developers, to see if they can uncover any issues themselves. It would also be possible to increase the accuracy of the flight path generation simulations by adding variables such as wind speed, but this would make the project take a significantly longer amount of time and is not feasible with the resources at hand.