***ILP Planning document***

# **Requirements to be considered**

In this document, I am going to consider only 3 of the requirements from section 1 and analyze them in detail in order to create a test plan analysis. The requirements I am going to consider are:

* **Requirement 1:** *“The system should ensure that the drone never enters a no-fly-zone”*.
* **Requirement 2:** “*The system should validate each order for the given date, and, in case some of the order information is not valid, label it with the appropriate order outcome depending on which information was invalid”*.
* **Requirement 3:** *“The system should be able to retrieve data about different components (such as location of the University’s Central Area, locations of No-Fly-Zones, restaurant locations and order information) from a given REST server”*.

The selected requirements vary in their respective levels: requirement 1 is a **system** level requirement, requirement 2 is an **integration** level requirement, and requirement 3 is a **unit** level requirement.

# **Priority and pre-requisites**

In this section, we consider each of the 3 requirements, and provide a short assessment of its T&A (Test and Analysis) needs.

1. **Requirement 1:**

* This is a safety requirement with implications for several stakeholders, so it will be a high priority to ensure we meet this requirement
* Therefore, we will need to devote a high level of resource to ensure the requirement is met
* Since the requirement is a high priority, we will need to consider at least two different T&A approaches. These approaches are going to be ***partition*** and ***redundancy***
* Although this requirement is a system level requirement, checks can be performed in order to detect bugs earlier in the development process. This might increase T&A effort in the short term but would result in less checks being performed at the later stages of development.
* We will also need to consider the possible inputs and outputs of the path finding algorithm:
* **Inputs:**
* A list of orders for the given date
* A starting location on the map (for the first order, this will be Appleton Tower)
* The remaining number of moves that the drone has
* **Outputs:**
* Can\_be\_delivered: a Boolean (True/ False) value that indicates whether or not the drone has enough moves left to deliver the order
* If can\_be\_delivered is True, then also a list of points constituting the flightpath the drone takes in order to make the delivery
* **Specification:**
* An order is only delivered if can\_be\_delivered is True and every point in the flightpath is not inside a no-fly-zone
* The partition principle suggests that to help ensure safety we should decompose the requirement into a ***functional component*** that ensures a drone’s path to a point on the map doesn’t cross a no-fly-zone, and a ***structural component*** that ensures the drone never enters a no-fly-zone when delivering orders for a day. Therefore, this principle suggests we get two different aspects that will need to be considered in the plan:
* Some sort of inspection of the path planning algorithm to ensure the requirement is properly implemented by specific code and that there is no extraneous code that cannot be seen as implementing the requirement.
* A later exhaustive test that checks several combinations of order inputs and checks the result conforms to the specification.
* The redundancy principle suggests that redundant checks can increase the capabilities of catching specific faults early or more efficiently. This relates to the point made in the partition principle, where this requirement is split into the two aspects mentioned above. These tasks can be further broken down into different types of checks being performed, such as ***validation checks*** for both tasks, ***static checks*** as part of the first task, and ***assertion properties*** as part of both tasks.
* These two principles suggest the following tasks should be added to the test plan:
* Generate synthetic data to test the flight path algorithm
* Building some sort of scaffolding to simulate the software and run the tests on the synthetic data earlier on
* Design more extensive test cases at system level to ensure the software works as expected on real data (from the REST server)

1. **Requirement 2:**

* This is a correctness requirement at integration level, so it will have a medium priority
* Therefore, a moderate level of resource (greater than for requirement 3, but less than for requirement 1) will need to be devoted to ensure the requirement is met
* Since the requirement is of medium priority, we will consider one of the T&A approaches, namely ***redundancy***
* The redundancy principle suggests that redundant checks can increase the capabilities of catching specific faults early or more efficiently. In the case of this requirement, we can use various ***validation/ verification checks***, as well as ***static checks*** on synthetically generated data, in order to ensure this requirement is met.
* We will also need to consider the possible inputs and outputs as part of this requirement:
* **Inputs:**
* Current\_order: the order object being considered
* Outcome: the outcome field for the current\_order object
* **Outputs:**
* is\_valid: a Boolean (True/ False) value that indicates whether or not current\_order is a valid order
* the outcome field being set to one of the 10 possible values (***ValidButNotDelivered, Invalid, Delivered, InvalidTotal, InvalidPizzaCount, InvalidPizzaNotDefined, InvalidCvv, InvalidExpiryDate, InvalidCardNumber, InvalidPizzaCombinationMultipleSuppliers***) according to whether or not is\_valid is True or False
* **Specification:**
* is\_valid is only set to True if all the validation checks pass, and the outcome field is set to ***ValidButNotDelivered***; otherwise, if an order is invalid, the outcome field is set to the appropriate invalid label
* These points made so far suggest the following tasks should be added to the test plan:
* Generate synthetic data to test the flight path algorithm
* Building some sort of scaffolding to simulate the software and run the tests on the synthetic data earlier on
* Design more extensive test cases in order to integrate the data obtained dynamically for the REST server with the order validation

1. **Requirement 3:**

* This is a liveness requirement tested at unit level, so it will have quite a low priority
* There are validation and verification checks that will need to be performed.
* Information about the different data from the REST server is needed (characteristics such as length of data input, as well as specific information relating to individual pieces of data (e.g., order details))
* To validate and verify this requirement, a simple unit test that checks for correct functionality should suffice
* The testing for this requirement can be done at any stage of the development process (either early on, or at a late stage, depending on the time and resources available to spend on the requirement)
* The following inputs and outputs need to be considered as part of this requirement:
* **Inputs:**
* **Central Area:** the set of co-ordinates that constitute the University’s Central Campus Area
* **No-Fly-Zones:** the list of all no-fly-zones that the drone should avoid when making deliveries
* **Orders:** the list of all orders available on the REST server, for all possible dates
* **Orders for a given date:** the list of all orders for a single specific date
* **Restaurants:** the list of all the restaurants available on the PizzaDronz app
* **Outputs:**
* **Central Area:** there are 4 corners
* **No-Fly-Zones:** there are 4 no-fly-zones
* **Orders:** there are 7050 orders in total
* **Orders for a given date:** there are 47 orders for any given date
* **Restaurants:** there are 4 restaurants in total

# **Scaffolding and instrumentation**

In this section, I am going to describe what scaffolding and implementation are needed in order to carry out the given tasks (and this may give result in more tasks to build scaffolding and instrumentation). For the selected requirements, we have the following:

1. **Requirement 1:**

* Some sort of simulator for the system, so that it’s possible to test the software. This is scaffolding and will need to be scheduled early.
* Having data for the simulator may involve some effort that needs to be
* scheduled
* The system test will combine the simulator with the actual data obtained from the REST server
* The previous tests (before system level) will combine the simulator with the synthetically generated data

1. **Requirement 2:**

* Some sort of simulator for the system, so that it’s possible to test the software. This is scaffolding and will need to be scheduled early.
* Having data for the simulator may involve some effort that needs to be
* scheduled
* The integration test will combine the simulator with the actual data obtained from the REST server

1. **Requirement 3:**

* This section will not require any scaffolding since a simple unit test will be enough to verify this requirement.
* Therefore, the only instrumentation needing to be scheduled is the implementation of the unit test

# **Process and Risk**

We can clearly see from the previous sections that designing the simulator and the synthetic data needs to be done at an early stage of development. This will also be quite a long task, and so significant time and effort should be assigned just for this task alone before moving on to other tasks. The integration test for requirement 2 can also be done before the final stages of development, but once the simulator has been set up and proven to work; designing the integration test can be done concurrently with development, since it is not as big of a task as the simulator. Lastly, considering the minimal time and effort needed to create the unit test, we could perhaps create the unit test in parallel with the integration test, since that will allow us to have more time at the end for system testing, and also give us more time to spot bugs in an earlier stage of development.

One of the key risks, related to ***technology risks***, is that of the quality of the system to be tested. For instance, if the system is very poorly designed, or the implementation is really difficult to understand, this may result in delays in the testing process at all levels (unit, integration, system). In addition, it would make debugging very difficult in case bugs are detected by the tests. Therefore, one of the main limitations of the plan is that it could fail to account for very poor design choices on the developers’ end. In the context of this project, this risk is mitigated since there is only one developer (myself) responsible for both the design and testing, and hence generating tests for a design I created myself would not be an issue.

Another key risk relates to the ***level of assurance*** necessary for some of the project requirements. In our case, for requirement 1, there is a very high level of assurance that is needed, since violating this requirement would defy the whole purpose of our system (by violating the needs of several stakeholders). This, in turn, gives rise to ***scheduling risks***, since underestimating the amount of time and resources needed to verify this requirement could have a large impact on the development of the software both timewise and quality-wise. The main way to mitigate this risk is to ensure our simulator for the software environment is highly adequate and functional, as this will ensure that the instrumentation used for testing will give us an accurate representation of whether or not the system works as expected. However, this may give rise to additional validation/ verification checks that will need to be accounted for in the test plan.

Aside from the two risks mentioned, there are also risks related to the actual instrumentation of the test plan. Some of the limitations I am going to consider below include whether or not ***the synthetic data is representative of the real data***, as well as whether or not ***the test suite adequately verifies the requirements***.

First, we will consider the risk regarding whether or notthe synthetic data is representative of the real data. This would obviously be a serious issue, since it could delay finding bugs to a very late stage in the development process. This would, in turn, slow down deployment, as well as other system tests that are dependent on the given feature. This risk is particularly relevant to requirements 1 and 2, since having a wrongful sense of assurance that the system is working (when in fact it is not) could be very problematic when integrating the remaining components. One way to mitigate this risk is to ensure the synthetic data is similar to the actual data being used; for instance, for requirement 2, this can be done by creating order objects with similar information to the ones from the REST server. In doing so, we more accurately mirror the real data, ensuring adequate testing and not gaining a false sense of assurance.

Now, let us consider the adequacy of the test suite. This risk is relevant for all 3 requirements; however, it can be more costly than the previous risk, especially if the test suite is not adequate for testing requirement 3. Since our system relies on getting its data dynamically from a REST server, having faulty data retrieval (that we are unaware of) can have catastrophic consequences for the whole system. In order to mitigate this risk, we can ensure we have enough test coverage, by testing all possible cases in order to ensure we verify this requirement. Whilst it can be quite costly timewise and resource-wise, in the long run it will result in a higher quality test suite, and also a higher quality end-product that we can be more certain that it works as expected.