**Test Plan Document**

**Requirements covered in the document:**

Considering the fact that there are a lot of functional requirements in the Software being tested, we will only consider a small subset of them, only three of the ones stated in the Software Requirements Document. Even though we are selecting a small subset, we ensure that all types of tests are covered, namely, integration, system, and unit. On top of this we also make sure that all three priorities (low, moderate, high) are covered. The requirements being considered taken from the software requirements documents are:

1. **Unit Test Requirement**: “The system should consider two locations that have a Euclidean distance of 0.00015 between to be ‘close’ (so if the drone is ‘close’ to a point, we consider the drone to be at the point). This requirement could be categorized under functionality.”
2. **Integration test Requirement**: “The system should validate each order for a given date of operation. In case the order is not valid, label it with the appropriate order outcome depending on which information was invalid. The orders can have one of 10 outcomes (2 of which depict a valid order).”
3. **System Test Requirement**: “The system ensures that the drone moves are not what could be resulting in an ‘illegal’ move. An illegal move would be if a drone flew into a ‘no-fly-zone’ or if once the drone enters the ‘central area’, it does not leave until it makes it delivery in Appleton tower. Another variation of an illegal move would be if the drone leaves the ‘central area’ it should not enter the central area again until it picks up the pizza.”

**Priority-level and pre-requisites for each of the Requirements:**

1. **The Unit Test Requirement:**
   1. This is a functional requirement, which should have a low priority in the development process.
   2. A mathematical formula for calculating Euclidean distance should be defined and implemented in the system.
   3. Validation and verification checks should be performed to ensure the correct calculation of Euclidean distance.
   4. A simple unit test that checks for the correct functionality of the calculation should be implemented to validate and verify the requirement.
   5. The testing for this requirement can be done at any stage of the development process, depending on the available time and resources.
   6. The following inputs and outputs need to be considered as part of this requirement:
      1. Inputs:
         1. Two locations with their coordinates (x1, y1) and (x2, y2)
      2. Outputs:
         1. Boolean value indicating whether the two locations are 'close' or not.
2. **The Integration Test Requirement:**
   1. As it is a requirement for correctness at integration level, then it will have a medium/moderate priority since it is a correctness requirement.
   2. For this reason, ensuring that the requirement is met will require a moderate level of resources (higher than for low-priority requirements, but lower than for high-priority requirements).
   3. Since the requirement has a moderate priority, we should strive to take one analysis and testing (A&T) approach into account. In our case, we designate this to be Redundancy.
   4. Validation and verification checks will need to be performed to ensure the requirement is met.
   5. The inputs and outputs for this requirement need to be considered:
      1. Input:
         1. An order object.
         2. Date of the order being considered.
      2. Output:
         1. Whether the current order is valid or not in the form of True/False.
         2. Set the outcome field of the current order object to one of the 10 possible values (2 of which depict a valid order). The invalid order is one of 8.
   6. The following tasks should be added to the test plan:
      1. Validate the order validation algorithm by generating test data.
      2. Test the order validation algorithm with a given order date by designing test cases.
      3. Ensure that the order outcome field is set to the appropriate value based on the validity of the order, on a case-by-case basis.
      4. Validate and verify the implemented solution to make sure it meets the requirements.
3. **System Test Requirement:**
   1. This would be done to ensure safety as well as minimizing obstruction to the stakeholders (students as well as the general public in the drone’s operational area/s).
   2. Ensuring the safety of the stakeholders will be of utmost importance so the requirement must be given a high priority to make sure that the requirement is met.
   3. Since it is a high priority requirement, we must consider the approach of partition and redundancy while performing analysis and testing.
   4. This requirement could be tested for bug detection early in development even though it is a system level test.
   5. According to the partition principle, we would have to partition the requirement into a functional and structural component in order to ensure safety. The functional part would ensure that the drone’s movement abides by the given constraints, i.e., not entering the no-fly-zones as well as abiding by the central area constraints. And structural part would ensure that the drone, in its flightpath for the day never violates the constraints.
   6. According to the redundancy principle, we would have to make redundant checks so that we can catch specific faults earlier and more efficiently.
   7. Consulting the Redundancy and Partitioning principles, we would be inclined to undertake tasks such as creating a synthetic flight path to test the path finding algorithm and building scaffolding to augment the working software.

**Instrumentation:**

This section will be used to describe the scaffolding and implementation needed in order to carry out the testing. Regarding the requirements being considered the scaffolding and instrumentation is:

1. **Unit test requirement:** The scaffolding for this requirement would be non-existent since this is just a unit test.
2. **Integration test requirement:**
   1. A simulation of the system so that the system can be tested. This scaffolding must be set up early. By setting up scaffolding early, the software can be tested more effectively, and any bugs or issues can be detected and resolved sooner. This helps in improving the overall quality of the software and in reducing the risk of errors in the software during operation.
   2. Obtaining and synthesizing the data for the simulation would require a lot of time as well as manual effort.
   3. The integration test will involve combining the simulation created for the system with actual data obtained from the REST server. This integration test is critical for ensuring that the software and the data being used are working correctly together. The test will verify that the system is able to validate each order for a given date of operation and label the order with the appropriate outcome depending on which information was invalid. The orders can have one of 10 possible outcomes, two of which depict a valid order. The integration test will be an essential step in verifying that the system is functioning as intended and that the software and the data being used are interacting correctly.
3. **System-level test requirement:**
   1. A simulation of the system so that the system can be tested. This scaffolding must be set up early. By setting up scaffolding early, the software can be tested more effectively, and any bugs or issues can be detected and resolved sooner. This helps in improving the overall quality of the software and in reducing the risk of errors in the software during operation. This helps in improving the overall quality of the software and in reducing the risk of errors in the software during operation.
   2. Obtaining and synthesizing the data for the simulation of the system would require a significant amount of time and manual effort.
   3. The system test involves combining the simulation with actual data obtained from the REST server to verify the functionality of the software and its interaction with the data. This system test is crucial in ensuring that the software can validate each order for a given date of operation and label it with the appropriate outcome, which can be one of the 10 possible outcomes with two depicting a valid order. The system test is an essential step in verifying that the software is functioning as intended and that it is working correctly with the data.

**Expected Risks**

1. Unit test requirement

The requirement: “The system should consider two locations that have a Euclidean distance of 0.00015 between to be ‘close’ (so if the drone is ‘close’ to a point, we consider the drone to be at the point). This requirement could be categorized under functionality.”

Calculating the distance between two points poses few risks, since an error would only occur if the points passed were in the wrong format or if the formula being used to calculate the distance was incorrect.

1. Integration test requirement

The requirement: “The system should validate each order for a given date of operation. In case the order is not valid, label it with the appropriate order outcome depending on which information was invalid. The orders can have one of 10 outcomes (2 of which depict a valid order).”

The requirement of validating each order for a given date of operation and labelling it with the appropriate order outcome if it is not valid, is crucial to the system. The validation process involves 10 possible outcomes, with 2 of them depicting a valid order. This process may take up to two days to complete.

To successfully implement this requirement, access to the REST server is necessary and the server should contain sample orders and precise information on the validation details. This requirement will be placed during the design and implementation stage of the SRET lifecycle process and will dictate the validation process the software should follow.

However, there are risks associated with this requirement. For instance, the REST server could have incomplete information, which could result in incorrect categorisation of orders. To mitigate these risks, it is important to have checks in place to ensure the data on the REST server is complete, as well as alternative sources or contingency plans in case the server fails or becomes unavailable. Additionally, the validation process could be overly complex or take up too much time and resources, causing delays and inefficiencies.

1. System-level test requirement

The requirement: “The system ensures that the drone moves are not what could be resulting in an ‘illegal’ move. An illegal move would be if a drone flew into a ‘no-fly-zone’ or if once the drone enters the ‘central area’, it does not leave until it makes it delivery in Appleton tower. Another variation of an illegal move would be if the drone leaves the ‘central area’ it should not enter the central area again until it picks up the pizza.”

An important aspect of the system is to make sure drone movements don't lead to illegal ones. Drones that fly into no-fly zones or enter the central area and leave without delivering to Appleton Tower would be illegal. When the drone leaves the central area, it shouldn't come back to the central area until it picks up the pizza from the restaurant.

There are, however, a bunch of risks involved with this requirement. The REST server has to be up to date on no-fly zones and the central area so that the drones don't fly in a way that could be considered as an illegal move.

Another huge risk of the system is that the flight generation could be extremely resource heavy, causing inefficient code along with delays.

The system also needs to be able to accurately track the drone's locations and make sure they are following their intended flight paths. There is a risk that drones may deviate from their planned routes and enter prohibited areas if there are any errors or glitches in the system.

It is also possible for the REST server to contain incomplete information, resulting in incorrect order categorization. The REST server must have checks in place to ensure that the data it contains is complete, as well as alternative sources or contingency plans in case of failure due to the REST server.

Lastly, validation can be complex and time-consuming, resulting in delays and inefficiencies.

**Evaluation**

By combining automated integration testing with real and synthetic data utilised at the unit level, the current instrumentation offers a thorough testing strategy. In order to give users confidence in the system's functionality and to find and correct any necessary flaws, the automated tests should provide sufficient coverage. To ensure that the automated tests are sufficiently comprehensive, coverage must be assessed with code analysis tools such as code coverage, mutation testing, and source control.

Manual testing, though, can help the system's robustness even more. With the use of this technique, the user experience would be more accurately simulated by recreating actual user behaviour. The manual testing procedure may have several drawbacks, such as the possibility that the developer conducting the testing is already familiar with the system's underlying mechanics and might not try all possible combinations. Additionally, manual testing needs more time and resources, which could be difficult given the limited time available. Despite these drawbacks, manual testing provides an in-depth understanding of the system’s overall behaviour and its response to real-life user behaviour. Therefore, manual testing is still important and necessary when it comes to testing applications.

At the moment, only actual REST servers are used for testing data retrieval procedures. Using a simulated REST server and simulated data for testing could make this better. Additionally, this would lessen the requirement for manually producing synthetic data. However, it would also take more time and money to put this plan into action.