LO 1

Analyse requirements to determine  
appropriate testing strategies

* 1. **Range of requirements: functional requirements, measurable quality attributes, qualitative requirements**

There are a multitude of user and system requirements for this system. I have categorised a range of these, spanning the various needs of the system such as drone movement and order processing.

1. Functional Requirements

* Drone must complete the entire flightpath within 2000 moves.
* Flightpath calculation must take less than 60 seconds.
* Drone must correctly select the optimal flightpath, which delivers the maximum number of pizzas within 2000 moves
* Drone must not cross any boundary of a no-fly zone
* Once the drone enters Central Area when returning to Appleton Tower, it cannot leave Central Area again before delivering the order.
* A customer should be able to place an order on the app.
* All orders should be correctly recorded in the database on the REST sever.
* Information from all orders should be correctly retrievable from the REST server.
* All recorded details of an order should be correctly validated:
* Payment card information
* Credit card number
* Expiry date
* CVV
* Customer’s name
* Number of pizzas in the order
* Source restaurant for all pizzas in the order
* Total cost of the order (compared to the restaurant’s menu prices)
* If an order is valid, the correct negative order outcome should be returned
* Valid orders should be considered for the final flightpath
* If a customer receives an order, it should be the correct order

1. Measurable Quality Attributes

* (Robustness) Drone must immediately terminate its flightpath when faced with inclement weather or illegal conditions, e.g. if it is pushed into a no-fly zone by strong wind.
* If required to terminate its flightpath, the drone should land in a safe place. (Where ‘safe’ can be defined depending on the regulatory code, such as 50m horizontal distance from any person[1](#_https://register-drones.caa.co.uk/d)
* System should be reliable, i.e., have a low Mean Time Between Failures (MTBF) and a low Mean Time to Repair (MTTR)
* The system should be easily maintainable, with clear coding standards and high modularity. It should have a low Mean Time to Change (MTTC).

1. Qualitative Requirements

* A customer’s personal information, including credit card information, should be stored securely and protected against unauthorised access.
* If an order is valid and deliverable, it should be received in a reasonable amount of time.
* When hovering, the drone should pick up and drop off pizzas efficiently, i.e. the endpoints of a delivery should be efficient, regardless of actual in-flight efficiency.
* Code should be manageable – well documented where necessary, well-commented, and written in a clear, concise way.
* System should be dependable – it should behave correctly in expected ways, and not fail in normal use.
  1. **Level of requirements: system, integration, unit.**

The category of system level requirements includes qualitative attributes. They require the holistic final function of the system to meet stakeholders’ needs. An example is the requirement to deliver orders within a reasonable amount of time, where ‘reasonable’ is entirely dependent on the particular customer’s opinion. A further example is how data security must be met at every sub-level of the system, and maintained in tandem across all modules. Any vulnerability on any level (such as poor access modifiers) will weaken security across the whole system. This is particularly key regarding data on the REST server; REST APIs are susceptible to injection and cross-site scripting (XSS) attacks, and so all input data must be carefully sanitised and validated to prevent failures in other modules.

Maintainability is also a systems level attribute. The system should be easily modifiable in order to correct faults and adapt to future features that provide increased functionality or performance.[2](#_https://software.ac.uk/resources/gu) Code quality should be high, with standardised formatting across the entire system. Comments and necessary documentation should be created throughout. Finally, having automated test coverage metrics is a good way of easily ensuring additional features are tested to a sufficient standard.

A further system requirement is dependability. The entire system must be reliable as a whole – it must deliver the expected service to users, with resilience through with redundancy checks and fault-tolerant code so that even when faults occur, there is a backbone functionality that can continue to run.[3](#_https://cs.ccsu.edu/~stan/classes/C) For example, if the drone is pushed into a no-fly zone by extreme wind, it should terminate its flightpath immediately to remain safe.

Regarding integration requirements, a major requirement is the integration of the application and the REST server where all data is stored for orders details, restaurants, and no-fly zone locations. It is imperative that data is seamlessly transferred to the application via the REST API without corruption or significant delay. Furthermore, the code that accesses the REST server URLs should be modular, and be easily expandable with any newly desired server endpoints for extra data tables.

Generally, subsystems should work in tandem. For example, the class that performs the A\* algorithm to determine the drone’s optimal flightpath should work smoothly with the class that determines whether a certain co-ordinate lies within a given polygon. That is to say, all potential drone moves the A\* algorithm selects must be validated efficiently to see whether or not they are in a no-fly zone. Only if they are valid can they then be added to the flightpath by the algorithm.

The unit requirements category includes all granular order validation functions, such as using the Luhn algorithm to confirm the checksum of a credit card number, and checking a card is not expired. These are independent functions and require simple test data.

Further unit level requirements are that the drone must not cross the boundaries of no-fly zones or illegally exit and re-enter the Central Area. Determining whether the drone’s current location is inside the polygon of a no-fly zone can be done with a ray casting algorithm; this calculation is independent of all other functions in the system.

* 1. **Identifying test approach for chosen attributes.**

An ideal attribute to test is the functional requirement that customer orders must be correctly validated. This means that the functions checking whether details are valid must only return true positive and negative results.

Validating an order includes checking the customer’s name, card details, number of pizzas in the order, source restaurant of the pizzas, and the total price of the order. This is a very wide scope, so my most thorough focus will be on the credit card validation checks. However, I will provide some tests for all functions, as such a wide scope can offer effective statistics on test coverage and valuable insight into the quality of the tests.

In order to test these functions, I will use functional, black-box unit testing. The test data will be systematically representative of certain valid inputs, boundary inputs, and extreme outliers. For each set of test data, I will compute the expected outcome, execute the test, and then compare the expected outcome with the actual outcome. The tests will be written in JUnit.

After this, I will use structural testing to test the control flow of the entire order validation class, and ensure that these methods are all called in the correct places. This can also provide statistical path coverage information.

The second chosen attribute is the component integration requirement that the flightpath module, when calculating the optimal flightpath, should regularly and efficiently test whether each potential next move would cross illegally into a no-fly zone. This requires communication with the LngLat class, which stores the longitude and latitude of a specific coordinate and contains a method checking whether that coordinate is within the boundaries of a specified polygon (a no-fly zone).

I will use black-box testing to ensure data has remained accurate after being processed and passed between the modules, and to ensure that invalid movements (where the drone would pass into a no-fly zone) were not added to the final flightpath.

Furthermore, one of the other flightpath requirements is that calculating the flightpath must take no more than 60 seconds. Therefore, instrumentation is key to test how efficient the integration is of these classes, including use of non-functional performance testing to analyse the LngLat algorithm computation times. If the algorithms in LngLat take too much time then the overall flightpath calculation time could easily fail the <60 seconds requirement.

* 1. **Assess the appropriateness of your chosen testing approach.**

Regarding the requirement of order validation, functional unit testing[4](#_https://www.guru99.com/functional-t) is the most appropriate choice. The validation functions are small, granular, and independent of other functions and modules. Functional testing is suited to revealing missing logic faults, which are likely in these functions where boundary data is often forgotten or miscategorised. It also helps build clear, specific error messages and ensures the correct ones are being returned in each case. If functional testing was not used enough here, it is inevitable that logic faults would be carried into integration and system testing, at which point it would be extremely difficult to trace the source of errors back to the order validation module.

Using structural coverage[5](#_https://www.softwaretestingo.com/st) afterwards ensures that not only have all paths through every function been tested thoroughly and work correctly, but that when an order is being validated, all validation methods are correctly called in sequence until an error occurs or until the order is deemed valid. It will help make clear that there is not one function regularly throwing a mistaken error and preventing the rest from being called.

The integration requirement of correct drone movement is best suited to component integration testing as the tests must ensure the components (LngLat and Flightpath classes) are interacting correctly and efficiently, without corrupting or losing data. As part of this, parameters and data units must match up. It is imperative to detect any issues that happen during integration as early as possible, as errors will propagate and could have a significant impact on the reliability of delivery data on the final flightpath, such as the drone heading to incorrect locations or taking inefficient or illegal paths. A poor flightpath will have a huge negative impact on overall stakeholder satisfaction.

Regarding performance testing, it may be useful to measure how long LngLat takes to determine whether a coordinate is inside a no-fly zone and how long the Flightpath class takes to calculate the flightpath if it ignores illegal moves into no-fly zones. Subtracting these could provide a useful metric of how much time is spent processing each move, and whether there are inefficiencies to be ironed out within LngLat.

**References**

## <https://register-drones.caa.co.uk/drone-code/where-you-can-fly>

## <https://software.ac.uk/resources/guides/developing-maintainable-software>

## <https://cs.ccsu.edu/~stan/classes/CS410/Notes16/10-SystemDependability.html>

## <https://www.guru99.com/functional-testing.html>

## [https://www.softwaretestingo.com/structural-and-functional-testing/](https://www.softwaretestingo.com/structural-and-functional-testing/%20)