LO2

# 2.1 Constructing the Test Plan

Requirements 1 & 2 specify that:

1. *The customer should receive their order and it must be correct.*
2. *The flight path must be generated in under 60 seconds.*

## R1 Testing Specification:

* **Customer Card Tests -** *cardNumberLengthTest, validCardNumberTest, cardCVVTest, cardExpiryTest*: These tests are to be implemented in order to verify and validify the payment method used by the customer. It also verifies that the current order is legitimate.
  + **Input:** Customer card details with the JSON Object containing their orders
  + **Output:** Test assertion fail or success if all the car details are correct.
* **Item Order Tests** **-** *notNullOrderItemsTest, notEmptyOrderItemsTest, numberOfOrderItemsTest, validOrderItemsTest, testOrderNumber*: These are implemented to verify the legitimacy of the individual items the customer is ordering. It also tests for the amount of order placed and makes sure the customer does not exceed the maximum number of pizzas being delivered by drone.
  + **Input:** A list of orders the customer has ordered with the JSON object equivalent.
  + **Output:** Test assertion fail or success if there are the correct number of valid orders being put through.
* **Final Validation Tests -** *testOrderPrice, customerTest*: To comepletey verify that an order comes through as a completely authentic. Both tests would finally verify that an order is valid and can be passed on.
  + **Input:** The total price of the orders and customer name as according to the JSON objects.
  + **Output:** Test assertion fail or success if price and name are correct.
* **Final Integration Test –** *fullOrderTest*: Finally, this test integrates many of the core parts of the system, form the drone functionality to the Battery’s durability. The full list of modules tested by this integration test are: *Battery, CentralArea, Drone, FlightDirection, FlightPoint, InvalidOrderException, LngLat, Menu, NoFlyZones, Order, ResponseHandler, Resturant, State*.
  + **Input:** The full order in JSON format that the customer has requested to be delivered.
  + **Output:** Test assertion fail or success if order outcome is marked as *delivered.*

## R2 testing specification:

For testing R2, multiple of the same performance tests were conducted in succession. The test had the same goal each time, drone needs to deliver the orders to the customer successfully and within a 60 second time window. The following were the Inputs and Outputs:

* **Inputs:** The items ordered by the customer along with any card details and costings
* **Outputs:** Test assertion fail or success if order outcome is marked as *delivered* and is delivered within 60 seconds.

# 2.2 Evaluating the Test

The test plan constructed above does a pretty good job at testing very basic functionality of the program. Much of the unit tests are set out to validate individual methods, while the final integration test for requirement 1 sets out to combine everything together to test how 90% of the program works when all modules are told to work together. Since the majority of the tests are only simple unit test, the plan has several flaws when it comes to requirement 1. One of these flaws includes the lack of System wide testing. It would be useful to fully test the system as a whole with multiple sets of inputs and outputs. This does not happen in the specified unit test.

For requirement 2, the single test specified is only supposed to evaluate the performance of the system. There is no need for many more tests since requirement 2 is just a timeframe the program must be completed by. You can easily evaluate this requirement with multiple similar performance tests such as the one described in the R2 testing specification.

# 2.3 Instrumentation of the Code

Since this was a relatively simple simulation, many of the coding instrumentations were easily researched and implemented. The most widely used instrumentation during the testing was the run-time environment. This allowed debugging which made the tests easier to understand and more transparent. Most modern IDEs provide some sort of debugging features. Breakpoints are also one of these features which are very commonly used. These can be placed within tests to further analyse fragments of code, making bugs easier to discover. Unit test libraries were also used.

# 2.4 Evaluation of Instrumentation

Most of the instrumentation used during testing was highly effective. For example, the debugging features such as breakpoints were instrumental in resolving any logical errors. Bugs were also quite common while testing was underway, however these were quickly resolved once tests were conducted. Most bugs were found due to the run-time environment catching exceptions, which were then translated and displayed. Each error was always narrowed down to a single line due to this useful tool.

The main downside of these kinds of complex debugging instrumentation is the speed. Testing via breakpoint can usually be quite slow since you have to go sometimes line by line until you find the problem. As a result, unit tests were always run before any debugging took place.