***ILP Testing Evidence document***

# **Testing techniques used**

Testing was performed at all 3 possible levels: ***unit***, ***integration,*** and ***system***. Examples of unit testing include test classes such as *TestOrder*, *TestRestaurant*, and so on; examples of integration testing include test class such as *OrderIntegrationTest*; the system test is in a class named *SystemTest*.

In terms of techniques being used, we can see the application of ***systematic functional testing*** in both the unit and integration test classes. The unit and integration tests were generated through ***specification-based testing***, which allows us to verify the functional requirements of the project related to the individual components, as well as integrating the real data from the REST server and checking that correct functionality is preserved.

Aside from this, we also see the use of ***structural testing*** techniques, such as ***equivalence partitioning***. This is clearly visible in the testing for order validation, since we need to account for several invalid labels (as well as a valid label) in order to ensure that the validation is working correctly, and the orders are sorted into the correct category according to the individual order details.

Another technique being used is that of ***boundary value analysis***. One example of where this can be seen is when testing edge cases for credit card details in order to ensure that the validation works as expected. Another example would be that of central area and no-fly-zones: checking edge case conditions, such as being exactly on the boundary for one of those areas, and ensuring the system makes the appropriate decision based on the requirements spec.

In order to assess how the testing measures the ***measurable attributes***, we will consider some of these attributes and how they were tested. For ***performance***, a timer was used to measure the system computation time and ensure that this falls within the required timeframe of 60 seconds. For ***functionality***, we considered all the requirements in the requirements specification, and how many of there were met after running the test suite. For ***fairness***, we considered all the orders delivered for a given date, and checked how many were delivered for each restaurant, in order to ensure that each restaurant had at least one order delivered.

# **Evaluation of adequacy of testing**

The testing that was performed is mainly ***optimistic***. For instance, if we consider the file writing (JSON and GeoJSON), these are tested at system level, but due to the large size of the files, we only perform automated checks to see if the files were created and do a manual check to see if the contents of the file appear to be correct. However, a malfunction of some sort on the REST server, the order validation mechanism, or the flight path planning algorithm could result in the contents of the files being incorrect/ faulty/ buggy. Another example of the testing being optimistic is in the case of the flight path planning algorithm: this is only checked during system testing, since we need real data to test the algorithm, and generating synthetic data would be too costly both timewise and resource-wise. Therefore, there could be missing cases that may not be considered and could have an impact on the reported level of functionality.

One evaluation criterion that can be used to motivate the choice of testing techniques is that of ***inadequate empty set***. This criterion states that “if a program is not tested at all, a tester cannot claim it has been adequately tested”. This is particularly relevant when considering the ***class coverage*** of our approach, which in our case, is 100% since our test suite covers all the classes in the program. This criterion clearly justifies the need for ***systematic functional testing***, since these tests ensure that we have tested our program and can therefore claim, with some degree of certainty, that it is functional and has been tested adequately.

Another criterion that can be used is that of ***non-exhaustive applicability***. This criterion states that “a tester does not need an exhaustive test set in order to adequately test a program”. This is particularly relevant when considering the ***test coverage*** of our approach, which in our case, is quite high. This is because our test suite covers all the classes in the program, and for both Enum classes, all the values have been accounted for.

Yet another criterion that can be used to motivate the choice of testing techniques is that of ***statement coverage***. This criterion states that “ensuring that the test set executed all statements in a program is a minimum coverage goal for a tester”. This is relevant for both ***class coverage*** and ***test coverage***.

The last two criteria justify the need for ***equivalence partitioning*** and ***boundary value analysis***. This is because, through these techniques, we are able to ensure that the system has been tested more thoroughly than just through the ***systematic functional testing*** and be more certain that the system works as expected and has been adequately tested. We can say so since these techniques cover more than just the basic functionality and explore other cases (including boundary conditions) that helps us better track and assess the behaviour of the system.

# **Results of testing**

One of the main issues encountered whilst testing the mentioned requirements was to do with ***order validation***. The issue here was that too many orders were labelled as invalid, and therefore the program was unable to work as expected. For instance, since too few orders were labelled as valid, sometimes all orders for a give restaurant would get lost; this had a huge impact on ***fairness***. This was resolved by creating an automated piece of code that looped over all orders on the REST server and printed out how many orders fell into each of the 10 possible categories. In this way, I was able to identify the source of the issue, which was my credit card validation, and fix the code accordingly.

Another issue detected whilst testing was related to the ***drone avoiding no-fly-zones***. This issue essentially consisted of the fact that the drone disregarded the no-fly-zones and went directly through them when making deliveries, which defied the whole purpose of the system and clearly impacted ***functionality***. This was resolved through debugging, with the source of the problem being that the no-fly-zones were never initialized, and so the drone had no way of realizing that no-fly-zones actually existed.

However, a follow-up issue to the previous issue was that, once the no-fly-zones were initialized, the runtime of the system was above the required “60 seconds or less”. This, of course, was related to the ***efficiency*** requirement, and severely impacted ***performance***. The actual issue here was that the program was stuck in an infinite loop whenever the drone got really close to a no-fly-zone. Due to the nature of the algorithm (a greedy algorithm) the drone would constantly fluctuate between the two same points on the map whenever it got close to one edge of one of the no-fly-zones. This issue was fixed by adding an additional if-statement to check if the position had already been visited, which ensured the system would never get stuck in an infinite loop, and also solved the runtime issue.

# **Evaluation of testing results**

The three criteria mentioned (***inadequate empty set***, ***non-exhaustive applicability***, ***statement coverage***) all relate to the concepts of ***class coverage*** and ***test coverage***.

The first concept (***class coverage***) ensures that all the classes in the system have been accounted for in the test suite. This means that the design of the system has been tested, and all components appear to function accordingly. Since all individual components function accordingly, this reduces the possibility of bugs/ faults arising when running the code, giving us more confidence that the testing is adequate, and the system works as expected.

The second concept (***test coverage***) ensures that a good proportion of the actual code is being executed and therefore tested. This means that the multiple scenarios/ cases are being accounted for and tested, and the behaviour of the system in these scenarios is noted and compared with the expected behaviour. In doing so, we can detect any faults in the code more easily, and therefore also correct these faults. By having a good test coverage, we can be sure that most possible outcomes are being considered, and, assuming the system works as expected in all scenarios, we can be more confident in our testing.