1.1 – Range of requirements

See Requirements document.

1.2 – Level of requirements

See Levels of Requirements document

1.3 – Test approaches

To approach testing for this system, I decided to split them into two parts: black box and white box testing. Due to the nature of this course and ILP, I only managed to start the testing phase when most of my system was finished. Thus, faults in the unit classes were not caught particularly early into development. If I had more resources (time, personnel, etc), the testing phase would have more accurately reflected a professional setting where unit classes are coded first, tested, then integrated with others.

For white box testing, I employed unit and integration testing. These two are necessary and valid approaches because I am the sole developer of this system as well as its tester, thus I have full knowledge of the implementation details and can change the internal structure of the code as needed.

On the other hand, examples of black box testing that I have used are functional testing, compatibility testing, and performance tests. Functional tests such as category partition tests were used to verify inputs and their corresponding outputs. Furthermore, I used compatibility testing in conjunction with performance tests to not only verify that the system ran efficiently, but that it ran efficiently on all operating systems.

1.4 – Appropriateness of chosen testing approach

Due to the lack of resources, as well as the nature of this system only existing as a back end, there were some limitations on the types of testing that I could use. One class of testing that I could not utilise was security testing, as I do not carry out transactions and/or store card details, while another, usability testing, was not possible as there was no front end to my system as well as not having any alpha/beta testers.

Although I have created tests that should cover most edge cases in general, there will be some that have not been found. This is due to not having more time to develop these tests, as well as not having enough data to test the performance in extreme cases. However, for something like the performance tests, it is unlikely for these situations to ever occur and thus the testing that I have done will be sufficient for now. Another limitation is that during compatibility testing, I did not have access to a Linux based system, so data for that operating system is incomplete.

2.1 – Construction of the test plan

See Test Planning document.

* 1. – Evaluation of the quality of the test plan

See Test Planning Document

2.3 – Instrumentation of the code

See Test Planning document.

2.4 – Evaluation of the instrumentation

For R1, I think that the input spaces that the driver generates could be too general. It could output a set of points which are not a feasible area in a real-life scenario. To combat this, I could limit the range of coordinates that the driver generates. Furthermore, I would be re-using the no-fly zones provided in the Informatics Large Practical coursework, which will be an adequate real-life example of such areas. However, this will limit the types of no-fly zones that will be used for testing, and thus there could be polygon shapes which break my system. One example of how this could occur is outlined in LO2.2. Using the JGraphT library, this will ensure that my algorithm is functioning correctly as it is an official Java library which other users also employ, and I can check that the outputs of the Dijkstra’s and A\* algorithms match that of mine.

Regarding R2, the logging instrumentation system could be more detailed, and it could log additional information about how much space the algorithm is taking up while calculating the shortest path, and I could optimise not only the runtime but the space complexity as well.

3.1 – Range of techniques

See LO3 Document for more detailed explanation.

* R1 uses category partition testing for the pathfinding algorithm.
* R2 uses category partition and performance testing to vary the OS.
* R3 uses structural and category partition testing for order validation.

3.2 – Evaluation criteria for the adequacy of the testing

For R1, the approach to the testing of this sub-requirement is pessimistic since there is one specific behaviour that do not demonstrate 100% freedom faults. The adequacy criteria that I chose for this requirement is a combination of specification-based criterion, defect density, and resources needed to code the test suites and fix the potential faults.

R2 also uses a simplified requirement, instead of finding the shortest path of moves, I now look for a set of nodes which has the shortest path to the destination. The criterion that I adhere by is the execution time of the whole system, making sure that it runs in an acceptable time. This is calculated by taking the milliseconds at the start of running a subsystem and subtracting the time when it has finished.

For R3, I used a specification-based criterion in combination with IntelliJ’s coverage feature, as this requirement naturally lends itself to trying out different types of values for each attribute of the order, and I can check that most if not all lines of code were executed. Moreover, I have simplified the process by not testing for all combinations of card values such that I only test for each error case once, and the AMEX/NAMEX (Non-AMEX) properties mean that I do not have to test for invalid combinations of card details.

3.3 – Results of testing

See Testing Log document.

3.4 – Evaluation of the results

See System Evaluation Activities document.

4.1 – Identifying gaps and omissions in the testing process

One way the testing process could have been made better is if there were more varieties of no-fly zones. Currently, I am using no-fly zones provided by the REST server, but if there was more time, I could have come up with a no-fly zone generator that produces an arbitrary number of no-fly zones with an arbitrary number of vertices. Thus, this would have raised any other unknown edge cases and made my system more complete.

For R2, I did not have access to other forms of operating systems such as a Linux-based one. This would have given me extra confidence that my system works on all operating systems, but since Linux is viewed as a faster and more efficient OS than Windows for example, I can assume that a similar if not better performance will be recorded on Linux.

Furthermore, I did not have access to a varied and more complicated dataset, which would have had a bigger impact on the computational resources required. When drawing the visibility graph to a GeoJSON map, I realised that the computation would not take very long even if there were 100 restaurants, since there would be at least one path which connects a no-fly zone vertex or restaurant to another restaurant going from Appleton Tower. Thus, the A-Star algorithm would not need to look beyond one or two edges at maximum to find the shortest path. The calculations would take longer only if the restaurants and no-fly zones were more spread out, such that there were fewer paths and the algorithm had to travel deeper into the graph.

There was a similar problem regarding R3, in that the dataset was not varied enough. Each day contained orders with the same errors in their outcomes. When running my software, it turned out that there were the same number of orders delivered each day. One solution for this could be to have a random order generator which chooses an OrderOutcome value and sets its attribute to match that outcome. However, due to time constraints, I was not able to come up with one myself.

4.2 – Identifying target coverage/performance levels for the different testing procedures

See System Evaluation Activities document.

4.3 – Discussing how the test carried out compares with the target levels

See System Evaluation Activities document.

4.4 – Discussion of what would be necessary to achieve the target levels

See System Evaluation Activities document

5.1 – Identify and apply review criteria to selected parts of the code and identify issues in the code

See Code Review Activities document

5.2 – Construct (Design) an appropriate CI pipeline for the software

See CI Pipeline Design document

5.3 – Automate some aspects of the testing

See CI Pipeline Design document

5.4 – Demonstrate the CI pipeline functions as expected

See CI Pipeline Design document