Gaps and Omissions in Testing:

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.

Evaluation Activities:

R1, R3: The system should be able to find the shortest path by avoiding no-fly zones/The system should be able to classify orders based on their attributes.

Class name	Target Coverage	Actual coverage (Calculated by IntelliJ)
LngLat	100%	83%
Polygon	100%	67%
Card	100%	92%
OrderChecker	100%	87%

The adequacy criteria for R1.1 (Coverage of LngLat and Polygon) is sufficient, as all lines in the methods which are relevant are executed at least once (excluding toString, hashcode, etc). The defect density is also low, only occurring in the case where three vertices are collinear, which produces an incorrect visibility graph.

However, the same cannot be said for R1.2. This relies on a third-party library JGraphT and a random number generator to produce restaurants.

- Limited no-fly zones
- Random generator could miss a class of coordinates which are more prone to errors

One way that some errors might not be caught for R3 is if in the real-world context where these card details need to be authenticated, is if the card does not exist. Therefore, we would need a way to check with the relevant card provider and check that it is indeed an existing card capable of transactions.

- Using category partition, I have extensively tested every combination possible, which ensures that this requirement is met.

R2: The system should take at most 60 seconds to find the shortest paths for all orders in each day.

System name	Target performance level	Actual performance level
Order validation (2023-03-25)	25ms	50ms
Visibility graph generation (10 – 20 restaurants)	5ms	10ms
A-Star search	0.5ms	0.5ms (Average)
Entire system	0.4 seconds	1.4 seconds (First run) 0.6 seconds (Subsequent)

By simplifying moves to nodes, there will be less comparisons between objects and thus the tests will not suffer from floating point errors. The performance criterion for R2 is adequate for the current dataset and is the only way that we can test this requirement directly. However, since I have no reliable process for generating more complicated data (Bigger orders, longer paths to restaurants), there could be certain configurations which do not have a passable execution time and thus fail to meet this requirement.

In the performance adequacy criterion that I used, we can see that the entire system takes 1.4 seconds to run for the first time, decreasing to around 0.6 for subsequent runs. This is partly due to IntelliJ's caching of data received from the REST server, which take up the bulk of the run-time enabling the next executions to have a drastically decreased performance level. The same behaviour occurred for all other performance tests with the initial run taking a lot longer than the subsequent executions. For example, the order validation took around 50-60ms on the first run but dropped to around 10ms on subsequent runs in the same execution window. I have put down the run-time for just the first execution since currently, the system pools all orders for a given day then calculates the relevant paths. However, in a real-life scenario, a much better way for the system to operate would be to handle each order as it comes in, so that the pizzas can be delivered in an adequate time. Not only would this be a better approach in terms of customer service, but this would also cut down on the performance of my whole system, as it would only need to process one order and find the optimal path to one restaurant. Thus, the average performance level would continue to drop while the system is running, and a more realistic execution time would be significantly lower.

<u>How to achieve target levels</u>:

R1, R2:
The target coverage levels have been met, although they are not at 100%, the missed methods are *toString* and *hashcode*, which we do not need to test. However, to get a stricter guarantee that the requirement is met, there could be a custom no-fly zone generator instead of using ones that have already been provided in the REST server. This generator would produce different types of no-fly zones – convex, concave, as well as other areas which have an irregular shape. However, due to resource constraints, it was not possible to develop a custom generator in the timeframe of one semester with just one person.

R3:

To achieve the target levels, I would need to be able to carry out further performance tests on the efficiency of my system to ensure that it performs as close as it can to the target level. For example, there could be more variety in the orders, such as having more restaurants and pizzas, expanding the types of cards I accept, and increasing the number of orders per day. By doing this, I could get an idea of the limit of my system and try to improve its performance in the subsystems where it took the longest to execute.

Furthermore, I would also need to improve the run-time of my visibility graph generation. Currently, it generates the graph with $O(n^3)$, where n is the number of nodes (Restaurants, vertices of no-fly zones, etc). However, there are some advanced algorithms that complete in $O(n^2 \log n)$, or even in some cases, O(n). For this project, these were too advanced for me to be able to both understand and implement within the given time frame. However, in a professional workplace with proper resources, both in terms of time and personnel, this would be an achievable goal which would allow this system to produce better performance levels and reach the target that I have set.