Testing R1 - The system should be able to find the shortest path by avoiding no-fly zones:

Testing this original requirement is complicated, since there are infinitely many coordinates that you can choose from to make the shortest path, so we will have to limit this to an arbitrary number of no-fly zones. We can break this down into two further parts: The drone does not fly through a no-fly zone, and path to the restaurant is the shortest path. Thus, we can simplify R1 into two parts:

- R1.1 – The drone does not fly into no-fly zones.
- R1.2 – The path to the location has the shortest distance.

If there were more resources available in terms of time and personnel, I would have tested R1 using a combination of catalogue-based and category-partition testing. Since LngLat uses *Double* values, which is the base class that deals with all the intersection and distance calculations, we could test with regards to a range, such as at the extreme longitude degrees (-180 to 180) and latitude degrees (-90 to 90).

Cases for R1.1

- There are no no-fly zones [1.1]
- No-fly zones overlap each other [1.2] [NA]
- There is one no-fly zone [1.3]
- There are multiple no-fly zones [1.4]

Cases for R1.2

- The location is in a no-fly zone [2.1] [NA]
- The location could be Appleton tower or sufficiently close (<0.00015 degrees) [2.2]
- It could be outside of the zone where the drone can operate in [2.3] [NA]
- The location is valid. [2.4]

However, we can disregard [1.2], [2.1] and [2.3] as they are not likely to appear and does not translate well to real-world scenarios and thus have been marked as [NA].

Testing R1.1 – The drone does not fly into no-fly zones:

To ensure that R1.1 is satisfied, I use a visibility graph so that all node-to-node paths do not cross the no-fly zone. Therefore, we must test that the visibility graph output is correct to meet this requirement, and since the initialisation of the graph uses the Polygon class, we need to perform unit testing for LngLat, and integration testing for Polygon and Graph. These tests are devised using a category partition approach, by varying the inputs for the tested methods.

Unit test - LngLat		
Method	Test Name	Test condition
createLngLatInBetween	createLngLatAtStartPoint	Decimal = 0
	createLngLatAtEndPoint	Decimal = 1
	createLngLatInBetweenPoints	0 < Decimal < 1
	createLngLatOutsideTwoPoints	Decimal > 1
distanceTo	distanceToAnotherLngLat	LngLat1 != LngLat2
	distanceToItself	LngLat1 = LngLat2
	distanceToNull	LngLat1 != null && LngLat2 = null
closeTo	closeToAnotherLngLat	LngLat1 != LngLat2 && LngLat1.distanceTo(LngLat2) < 0.00015

	notCloseToAnotherLngLat	LngLat1 != LngLat2 && LngLat1.distanceTo(LngLat2) >= 0.00015
	closeToNull	LngLat1 != null && LngLat2 = null
	closeToAccuracyError	LngLat1 != LngLat2 && LngLat1.distanceTo(LngLat2) < 0.00015 + error
nextPosition	nextPositionNull	Direction = null
	nextPositionNotNull	Direction != null

Integration test - Polygon		
Method	Test name	Condition
calcCollinear	pointsCollinear	Three points have collinearity
	pointsCollinearOneInfinityLng	Three points have collinearity with one point having *inf* longitude
	pointsCollinearOneInfinityLat	Three points have collinearity with one point having *inf* latitude
	pointsNotCollinearClockwise	Three points do not have collinearity, and L_2->L_3 lies clockwise to L1->L_2
	pointsNotCollinearAnticlockwise	Three points do not have collinearity, and L_2->L_3 lies anticlockwise to L1->L_2
areLinesIntersecting	linesNotIntersecting	Two lines do not intersect
	linesIntersectingTwoSamePoints	The lines share the same endpoint
	linesIntersecting	Two lines intersect
areLinesIntersectingNonCollinear	linesNotIntersectingNonCollinear	Two lines do not intersect

	linesIntersectingTwoSamePointsNonCollinear	Two lines share the same endpoint but does not intersect
	linesIntersectingNonCollinear	Two lines intersect
isInsidePolygon (allowBoundaries = false)	isInsideNotAllowBoundaries	The point is inside the polygon
	isInsideOnEdgeNotAllowBoundaries	The point is on the edge and being on the boundary is counted as being inside the polygon
	isOutsideNotAllowBoundaries	The point is outside the polygon
(allowBoundaries = true)	isInsideAllowBoundaries	The point is inside the polygon
	isOutsideOnEdgeAllowBoundaries	The point is on the edge and being on the boundary is counted as being outside the polygon
	isOutsideAllowBoundaries	The point is outside the polygon
isLineIntersectingNfz	lineNotIntersectingPolygon	The line lies outside the polygon and does not intersect it
	lineOnVertexNotIntersectingPolygon	A vertex of the polygon is collinear with the line, but the line does not intersect it
	lineIntersectingPolygon	The line intersects the polygon
	lineInsidePolygon	The line is equal to the diagonal

		connecting opposite vertices in the polygon, and thus is intersecting it

Testing R1.2 - The path to the location has the shortest distance:

I have simplified this requirement such that it now reads: The list of vertices provided by the algorithm will give the shortest distance path to the restaurant. This is beneficial for testing since we now do not need to consider an infinite number of paths due to the continuous nature of a 2D space. By using the fact that R1.1 along with its units LngLat and Polygon have been thoroughly tested, if we can verify that the pathfinding algorithm is valid in the Graph class, this requirement will be satisfied.

The main driver I used to test this was the JGraphT library, which allows us to generate graphs perform A-Star search, which is the algorithm that I also use. If the output from this driver matches that of my custom graph, then I will be able to satisfy this requirement.

Furthermore, using this library, I can also confirm that my visibility graph generation is also sound and complete by writing the output to a GeoJSON map. I have also used a random coordinate generator to place pseudo-restaurants around the map which frees me from the limitation of the REST server only having four restaurants in total. This allows me to test an arbitrary number of restaurants, and in this case, my test involves finding optimal routes to 20 different restaurants.

R1 - Evaluation Criteria:

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.

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

R2 is a measurable attribute, needing the system to take at most 60 seconds to find paths for all orders on that day. I used a combinatorial testing approach for this by varying the OS used to test for this requirement. The OS used were Windows 10 and macOS, although if I had access to others, I could also have used Windows 11 and Linux. Instrumentation wise, I would need a way to measure how long the algorithm takes to run. Although a logging system would have been helpful as stated in the planning document, due to the nature of the coursework, I had finished my system design before starting LO3. Therefore, I am using the built in Java Clock class to measure how long the algorithm takes to calculate each move between node to node instead of an external logging system.

I tested how long the system takes to run by taking an average of the run-time of multiple days – 7 days, 10 days, 12 days, 30 days, and all days. No scaffolding was necessary here, as I needed a complete system to test for this requirement in the first place, and the data is provided through the ILP REST server on this URL - https://ilp-rest.azurewebsites.net//orders. Furthermore, I tested how

long the visibility graph generation took, and drew a graph to show that my implementation of the graph structure did not differ much in computation time when compared to the JGraphT Java library.

R2 - Evaluation Criteria:

R2 also uses a simplified requirement, instead of finding the shortest path of moves, I 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.

Testing R3 - The system should be able to classify orders based on their attributes:

For R3, I used two combinatorial methods to verify the different inputs of card and order details and their respective outcomes such that only valid inputs saw orders be classified as valid but not delivered.

A pairwise method was necessary to cut down the number of total tests since a Card object can take numerous values for each of its attributes and has additional constraints on them depending on the card type. Meanwhile, although the Order class can have different values based on its attributes such as order number, order items, and total price, I only need to check the validity of one attribute. If multiple attributes are invalid, it does not necessarily matter which outcome is attached as long as it is not valid. If this system was being worked on by multiple people, extensive testing would be expected, but since I am only one-person, additional constraints denoted by [single] was necessary.

Categories for Card:
[AMEX] = American Express
[NAMEX] = Non-American Express

CVV Length
- 3 [if NAMEX]
- 4 [if AMEX]
- Other [error]

CVV Value
- Numeric
- Contains non-numeric [error]

Expiry Date
- Date after current date [error]
- Date same as current date [single]
- Date before current date [single]
- Invalid date [error]

Card number length:
- 15 [if AMEX]
- 16 [if NAMEX]
- Other [error]

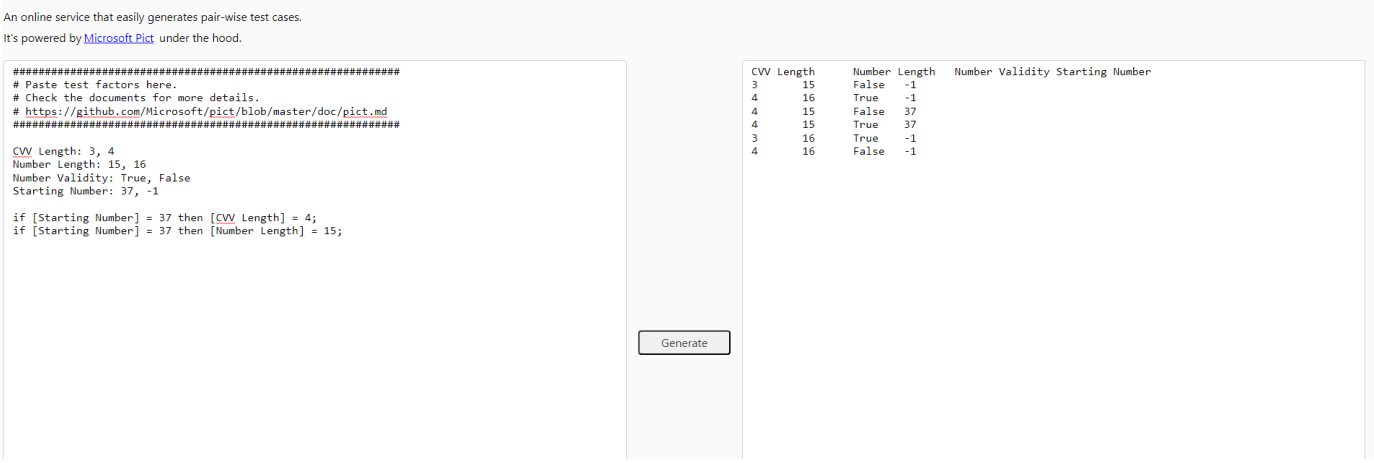
Card starting number:
- 37 [if AMEX]

- 0 [error]
- Other [NAMEX]

Card number validity:

- Passes Luhn's algorithm
- Does not pass Luhn's algorithm

Pairwise Combination Generator:



CVV Length	Credit Card Number Length	Credit Card Number Validity	Credit Card Starting Number
3	15	False	Other
4	16	True	Other
4	15	False	37
4	15	True	37
3	16	True	Other
4	16	False	Other

Categories for Order:

Order number:

- Hexadecimal 8-digit [single]
- Non-hexadecimal [error]

Card:

- Valid card details [single]
- Invalid cvv [error]
- Invalid card number [error]
- Invalid expiry date [error]

Order cost:

- Matches actual price plus delivery fee [single]
- Does not match [error]

Pizza count:

- 0 [error]

- > 4 [error]
- Between 1 and 4

Source of pizza:
- All pizzas from one restaurant [single]
- Pizza not defined in any restaurants [error]
- Pizzas from multiple restaurants [error]

R3 – Evaluation Criteria:

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. Furthermore, the AMEX/NAMEX (Non-AMEX) properties mean that I do not have to test for invalid combinations of card details, such as an AMEX card having a 16-digit credit card number (They have 15).