Learning Outcome 1: Analyse requirements to determine appropriate testing strategies

1.1 Range of requirements, functional requirements, measurable quality attributes, qualitative requirements, and more.

Functional	**Requirement**	
1	System must return the system creators student id	
2	Calculate Euclidean distance between 2 points	
3	Determine if 2 points are within 0.00015 of each other, returns true or false	
4	Calculate the next position given a position and angle and return the next position	
5	Verify if a point is in a closed region. Return true or false.	
6	Validate order and return its status. If an order is invalid it should return a message explaining why	
7	The system must be able to fetch data from external APIs for OrderValidation and delivery path calculation.	
8	Input validation to verify orders from the API are valid	
9	path validation to ensure paths don't enter restricted areas	
10	The system must calculate a path for a valid order	
12	Return a valid geojson path for compatibility with mapping tools	
Non functional	**Requirement**	**Type**
1	The system must be modular in case extra requirements are added	Scalable
2	The system must handle specific invalid orders, returning the correct error code without crashing	Robust
3	The system must return a valid path in less than 60 seconds	Performance
4	The paths must be optimal and use the least amount of moves possible to get to appleton	Performance
5	The system must be runnable through a docker image	Portable

---

1.2 Level of requirements: system, integration, unit.

To meet the outlined requirements, it was essential that:

System-Level Requirements: The entire system functioned to deliver the expected outputs for any valid input. The integration of various components, such as order validation and path calculation, was essential.

Integration-Level Requirements: Each unit within the system needed to integrate with others. For example, the validation module had to provide error-free data to the path calculation module, ensuring that downstream processes were not adversely affected by upstream errors.

Unit-Level Requirements: Each individual method and class was required to function correctly in isolation. For instance, the validation logic for orders were independently unit-tested to ensure accuracy.

These levels collectively ensured that the system met both functional and qualitative requirements while remaining robust against potential issues.

1.3 Identifying test approaches for chosen attributes.

Since we have multiple unit, integration and system level requirements we must also do unit, integration and system tests.

Unit Testing should be conducted to verify the functionality of isolated methods, such as validateOrder, which analyse and validate order data, ensuring that it meets the expected requirements to handle errors and edge cases.

Integration Testing will be done using Tools like Postman test all the system's endpoints, ensuring they correctly fetch API data and handle requests as expected. For instance, Validation endpoints would be tested to ensure they returned appropriate responses and are fetching data from the APIs correctly.

System testing will be done using Docker to simulate production-like environments, enabling comprehensive system level testing to ensure the system works.

1.4 Assess the appropriateness of the chosen testing approach.

The purpose of these test approaches is to verify that as much of the system works as it should be. By executing these tests, the system should work correctly according to the functional and non-functional requirements.

Unit Testing: Unit testing was effective for validating orders since the validation logic could be independently verified using a known set of inputs. The endpoint https://ilp-rest-2024.azurewebsites.net/orders provided a list of test orders, and its validity, simplifying the process.

Integration Testing: Postman was used extensively for integration testing. While it was straightforward to directly test most endpoints, testing calcDeliveryPathasGeoJson required additional effort. The only way to validate paths was by pasting the generated results into an external tool, GeoJSON.io, as the paths themselves lacked a built-in visual validation mechanism.

Runtime Analysis: Postman was used to measure the runtime of calcDeliveryPath and calcDeliveryPathasGeoJson. This provided valuable insights into performance which was useful to achieve one of the systems non-functional requirements.

Limitations:

There was no feasible way to test calcDeliveryPath by itself since there was no way to see if outputs were avoiding no fly zones and complying to the other restrictions. The solution to test calcDeliveryPath was to verify it using calcDeliveryPathasGeoJson since that was testable and the only difference was their output structure.