Learning Outcome 1: Analysing Requirements to Determine Appropriate Testing Strategies

1.1 Application Requirements:

Functional Requirements:

ID	Requirement
F1	The system *must* be able to validate a provided order, based on a variety of criteria. If the order is invalid, the system must return an error stating why. (The list of potential errors is found here).
F2	The system *must* be able to take valid orders and output a valid path from the restaurant to Appleton Tower.
F3	The system *must* be able to provide the path in a valid *GeoJSON* format.
F4	The system *must* return an error if the order is invalid.
F5	The provided path *must* not enter a *"No-Fly-Zone"* or leave the central campus once entered.
F6	The system *must* be able to take the Euclidian distance between two provided points
F7	The system *must* be able to return if two provided points are within *0.0015* degrees of each other.
F8	The system *must* be able to return if a point is within a region.
F9	The system *must* be able to return the next step in a drone's path if given starting coordinates and a direction.
F10	The system *must* be able to return the creator's student ID.
F11	The system *must* have basic *"isAlive"* checks.
F12	For all of the functional requirements above, the system *must* be able to return if a user's request is even valid. Reasons for invalidity include (but not limited to) malformed request, missing required parameter, non-existing endpoint etc.

Non-Functional Requirements:

ID	Requirement	Requirement Type
NF1	The system must return a valid path in less than 60 seconds	Performance
NF2	The system must gracefully handle specific invalid orders, returning appropriate error codes when necessary. It must not stop operating.	Robustness
NF3	The APIs must follow RESTful principals.	Technical
NF4	The system must be runnable on a Docker image.	Portability
NF5	The system must be built with Java using specific libraries and dependencies.	Technical
NF6	The system must dynamically pull information from ILP's dedicated server.	Interface
NF7	The system must be designed and implemented in a scalable manner.	Scalability
NF8	The system must be able to handle TLS (i.e. HTTPS) connections.	Security

1.2 Level of Requirements:

Requirement Level	Relevant Requirements
Unit	F3, F4, F5, F6, F7, F8, F10, F11
Integration	F1, F2, NF2, NF4, NF6, NF8
System	NF1, NF3, NF7

1.3 Identify Test Approach:

We will be utilising the entire testing pyramid to be ensure maximum test coverage of our application. Since we have multiple Unit, Integration and System requirements, we will need to conduct unit-, integration- and system-tests.

Unit tests (functional tests, error handling tests, edge-case testing etc) verify that individual units of code work in isolation. It helps catch any bugs before they're passed onto any other part of the system (e.g. the distance calculator works, next position function etc.).

Integration tests (microservice-to-microservice tests, API integration tests etc.) verify that individual components or modules work correctly together and exchange data and information as expected. For example, our order validation must query ILP's REST server to get a list of valid restaurants, before being able to verify other information.

Evaluate the entire system's behaviour in a real-world scenario, testing it as a unified whole. For PizzaDronz, this would look like (simulating) supplying a realistic request, and verifying the output is as it should be.

1.4 Assess Chosen Testing Strategy:

The goal of the testing pyramid is to create a balanced testing strategy that thoroughly tests the software at all levels. By covering the entire pyramid with the tests I suggested, all (functional and non-functional) requirements are adhered to by the system.

Unit tests are crucial for ensuring individual core blocks of code functions. They also prevent smaller bugs from compounding and becoming larger problems downstream. Since there's many moving parts in PizzaDronz, ensuring that they, individually, are all as intended, is crucial.

Integration tests ensure that collections of individual units work cohesively together. These are perfect for ensuring that the communication between the many moving parts is as expected. PizzaDronz' core functionality revolves around multiple different systems operating in unison. As such, testing they're all intercommunicable is critical.

And system testing allows us to confirm that our entire structure operates as it should. That said, the methodology isn't without it's flaws:

- ***Over-reliance on Static Test Data:*** The ILP course organiser helpfully provided a collection of static test data for us to verify our code works. It was limited and gave no formal guarantee that it would cover the entire range of possible errors. Creating more synthetic data would help cover all possible edge cases that verify the system is working as intended.

- ***Over-reliance on ILP Server:*** For collecting restaurant and coordinate data, I exclusively puled from the ILP server. We were formally guaranteed that the server would return different data during test time (vs. during the time we were given to build it). As such, if the data returned during test-time is drastically different, my program may not handle it appropriately. More testing (of a simulated server) would be beneficial.