Learning Outcome 2: Comprehensive Test Plan

2.1 Construction of Test Plan:

Throughout the project, I used employed Test-Driven-Development (TDD). This is when you create tests, then write its corresponding logic code, and then optimise the code without breaking the tests. This is beneficial as it allows us to catch bugs as soon as they're introduced, encourages simpler, cleaner code and reduces debugging time (a crucial factor given the limited available development time).

The requirements in this page refer to those defined in *LearningOutcome1.pdf*.

The Use of Manual Testing

Not all tests were performed programmatically. There was functionality in the program that could be checked trivially (e.g. visually), and implementing unit tests created unnecessary overhead in an already time-constrained environment. As such, some functional and non-functional requirements underwent Manual Testing. These include **F3**, **F11**. For non-functional requirements, they included: **NF3**, **NF4**, **NF5**, **NF6**, **NF7**, **NF8**.

F3 is a prime example. The goal was to output a GeoJSON-formatted string, which was easily verified by copy and pasting it and visually inspecting the path on *geojson.io*.

Test Plan:

ID	What Will Be Tested	How Testing Will Be Done	When Tests Will Fail
F1	Order validation logic including: *credit card validation*, *pizza counts*, *restaurant validation*, *price calculations*	*TDD* will be combined with *Data-Driven Development,* combining their strengths. Will test using 600 pre-defined test cases (300 valid, 300 invalid orders) through **system tests**.	- When validation returns wrong error code - When valid orders are marked invalid - When invalid orders are marked valid - When system crashes during validation
F2	Path generation from restaurant to Appleton Tower	**Unit tests** on core drone navigation functions: `*distanceTo*`, `*isInRegion*`, `*nextPosition*`, `*isCloseTo*`	- When path doesn't reach destination - When path calculations are incorrect - When navigation functions return incorrect values
F5	No-fly zone avoidance and campus boundary adherence	**Unit tests** checking path coordinates against defined no-fly zones and campus boundaries	- When path intersects no-fly zone - When path exits central campus - When boundary checking functions fail

F6	Euclidean distance calculation between points	**Unit tests** with known coordinate pairs and expected distances	- When calculated distance differs from expected - When function fails to handle edge cases (same points, null values) - When precision requirements aren't met
NF1	Path calculation performance	**Integration tests** measuring execution time for complete order-to-path generation	- When path calculation exceeds 60 seconds - When system timeouts occur - When performance degrades under load
NF2	Error handling and system stability	**System-wide tests** throwing invalid inputs at all endpoints	- When system crashes on invalid input - When incorrect error codes are returned - When error handling is inconsistent - When system state becomes corrupted

2.2 Evaluation of Test Plan:

My test plan is appropriate and lays a solid foundation, covering the essentials of basic functionality validation. Its key merits are:

- ***Comprehensive Coverage for Descending Priority:*** The two biggest priority requirements (**F1**, **F2**) are thoroughly enveloped with test coverage. Since this was a time-sensitive project, analysing the most important requirements and spending asymmetrical amount of time was critical. As such, the most important requirements received the most attention.

- ***Clear Failure Conditions:*** The failure conditions of every test are clear and concise. When it came to implementing the tests, the code was intuitive, as I had laid out the groundwork in English already.

- ***Good Utilisation of Testing Pyramid:*** In my tests, I utilise unit, system and integration tests. By combining all levels of the testing pyramid, I ensure maximum code coverage.

However, it's not without its faults. For example:

- ***Over Reliance on Unit Tests:*** The system as a whole covers unit tests extensively but has limited integration and system tests. This leaves a wide testing pyramid base but limited as it moves up. This gives a false sense of security, as often the realistic workflows are not getting tested.

2.3 Instrumentation:

To best test the project, I needed to implement an auxiliary framework of infrastructure code. This took the form of test scaffolding, performance monitoring, synthetic data generation and reporting.

Test Scaffolding:

I successfully implemented the mocking HTTP Requests with MockMvc, as well as Spring's MockBean for service mocking. This allowed testing the web endpoints without full application context. Additionally, I mocked the RestTemplate to avoid real HTTP calls. This helped me programmatically make any calls during the test-time, helping identify where any faults lay.

Instrumentation, Data and Reports:

Synthetic data was generated to ensure appropriate verification of edge-cases. Over 600 entries of potential orders, many with slight errors, were input into a JSON. From there, I created a pipeline that would get a list of predefined restaurants and valid order types. From there, it would run the program on every single order, using the order types as reference.

This pipeline also returned a report that detailed every test's result, their performance and the exact test a specific error occurred.

2.4 Evaluation of Instrumentation:

Strengths:

The instrumentation had many merits in its design and implementation. These include:

- ***Strong Synthetic Data:*** From the order validation dataset to restaurant data, there was a wide variety of real-world scenarios that were integrated and implemented in the JSON files. In fact, every possible combination of incorrect order was created.

- ***Strong Mocking:*** Strong isolation of components using Spring's mocking capabilities. This reduced external dependencies and allowed a controlled testing environment, where something specific could be tested without concerning ourselves with how it aligned with the rest of the system.

- ***Reporting:*** There was a sense of assurance after performing the tests, thanks to the confidence the reports provided. Knowing the test results, the code coverage they provided, and performance provided a level of confidence in the tests and how representative of the system they were.

Weaknesses:

The instrumentation was not without its faults, however. These included:

- ***Network Performance:*** There was limited evaluation on the network performance that the system endured.

- ***Load Testing:*** Not enough high load was put on this system for testing, as could be expected at peak hours (e.g. lunch). Better load testing would ensure reliability during these peak times.