Evaluation of Test Approach

In the following document I will assess the appropriateness of my testing approach, linking my justifications directly to the requirements discussed in **Range of Requirements** and **Level of Requirements**.

The requirements that fell under unit testing were the methods are focused on geometric behaviour such as fixed-distance movement and numerical threshold comparisons, which can be evaluated in isolation from the REST layer and external data. They shouldn't be tested at a higher level as it will make numerical errors harder to find. For example, should a failure in the arePositionsClose method occur, it could lead to an infinite loop during flight path generation which would be much harder to diagnose at integration or system level testing. Unit tests therefore provide the most precise and efficient validation for these requirements.

Integration testing was used to verify the behaviour of the REST controller and its interaction with validation logic and supporting services. Tests were executed using a Spring test context and exercised the /validateOrder, /calcDeliveryPath, and /calcDeliveryPathAsGeoJson endpoints.

Order Validation requires coordinated behaviour across several domain classes, such as orders, restaurants, pricing rules, and date-based constraints. Only when they are tested via the REST interface, which carries out validation and generates structured error responses, can their accuracy be meaningfully observed. Boundary, negative, and specification-based testing can be applied practically while maintaining automation and repeatability thanks to integration testing. Unit testing wouldn't represent real system behaviour and would need a lot of mocking.

Each validation rule was tested in isolation, with individual integration tests designed to trigger exactly one failure condition, reflecting the behaviour of the ILP auto-marker. Tests were implemented for all defined validation codes, including card validation, pizza count limits, restaurant consistency, and total price calculation including the fixed delivery fee.

Tests regarding the navigational requirements of the drone involve interactions between geometric calculations, region definitions, and routing logic, making them unsuitable for isolated unit testing. Integration testing is necessary to verify that movement constraints are correctly enforced when components are combined. This level of testing enables validation of spatial constraints and failure handling without requiring full end-to-end deployment.

System testing was performed using **Docker**, a software platform that allows you to build, test, and deploy applications quickly (Docker, n.d.). Through this I was able to validate full end-to-end behaviour as specified by the ILP coursework.

Tests regarding HTTP handling and inspection of GeoJSON flight path outputs are most suited for system testing. These requirements describe end-to-end system behaviour that emerges only when the service is deployed and exercised as a whole. System testing is appropriate because correctness depends on the combined operation of validation logic,

routing algorithms, REST endpoints, and data serialisation. Certain spatial properties of generated flight paths are difficult to verify exhaustively through automated assertions and therefore benefited from manual inspection.

Due to time constraints, system testing was partly manual; however, this was sufficient to establish specification compliance and identify integration-level defects not visible in lower-level tests.

Bibliography

Docker. (n.d.). Retrieved from https://www.docker.com/