LO1 - Software Testing

B196275

January 23, 2024

1 Requirements Analysis

I will perform a requirements analysis, which takes users' needs and defines the problem we are solving to derive my functional and qualitative requirements for this software.

Problem: The University Of Edinburgh (UofE) reached out to me to build them a back-end system to manage pizza deliveries for their new drone-based delivery system.

Client Requests:

- The UofE stated that they wanted the application to be efficient, and wanted delivery paths to be created within 60 seconds as delays could cause them to lose customers.
- They also said that orders must be valid before being delivered, as invalid orders being delivered is a waste of resources, and costs them money.
- They want data for the application to be retrieved from the REST server, and they emphasised the importance of the software integrating with this properly to avoid delays.
- They want the application to be relatively space efficient, however, they are not concerned about this as much and are happy to improve space efficiency in further iterations if required.

The client has also provided me with a detailed specification `ilpspec.pdf` with further information that can be used to aid when choosing what features to prioritise in development.

2 Requirements

The requirements for the PizzaDronz application can be broken down into two categories: Functional and Non-Functional. FR2, FR3, and FR5 have unit-level sub-requirements defined.

2.1 Functional Requirements

FR1: The application shall generate validate orders, generate flight-paths for valid ones, and output a deliveries-YYYY-MM-DD.json, flightpath-YYYY-MM-DD.json, and drone-YYYY-MM-DD.geojson files. [**System Level**]

FR2: The application shall accept only 2 valid arguments. [**System Level**]

1. A date of ISO 8601 format [**Unit Level**]
2. A URL of the correct format [**Unit Level**]

FR3: The application shall correctly validate orders following the criterion defined in the specification (refer to `ilpspec.pdf`). [**Unit Level**]

FR4: The application shall retrieve no-fly zones, orders, the central area, and restaurants from the rest server (`https://ilp-rest.azurewebsites.net/`). [**Integration Level**]

FR5 Flight-paths generated by the path-finding algorithm shall be valid. [**Integration Level**]

1. Flightpaths shall never enter no-fly zones. [**Unit Level**]

2. For paths starting in the central area, the drone shall not re-enter the central area after exiting it. For paths starting outside the central area, the drone shall not exit the central area after entering it. [**Integration Level**]

FR6 - The application shall generate flight paths for only valid orders. [**Integration Level**]

2.2 Qualitative Requirements

Each requirement has a quality attribute associated with it (the subheadings):
Performance:

NFR1: The application should generate all three files within 60 seconds. [**System-Level/Statistical Level**]

NFR2: The application should always use less than 2GB of system memory. [**System-Level/Statistical Level**]

Robustness:

NFR3: The application should exit gracefully rather than throwing exceptions. [**System-Level/Robustness Level**]

Maintainability:

NFR4: Each class and test should have a JavaDoc comment. [**System-Level/Maintainability Level**]

NFR5: Code should be written in a readable fashion. [**System-Level/Maintainability Level**]

3 Test Approach

Unit-level testing will be performed first, then any integration-level tests, and then any system-level tests if time permits. This is also known as a bottom-up approach.

FR1: I would use system-level testing to determine if the system outputs the 3 required files for differing inputs. We would provide the date and URL, and check that the correct files are generated.

FR2: I would perform unit-level tests to verify each criterion in the sub-criteria, to ensure the system handles the arguments properly. I would provide different values for the date, and different values for the URL to see how the application handles them.

FR3: I would perform unit-level tests to verify that every validation code is invoked correctly. I would try different order inputs and verify that it's validation code matches the expected validation code.

FR4: I would perform unit-level tests to verify that each method for each endpoint correctly handles certain exceptions and returns the correct entity. Further, I would perform integration-level tests to verify that the system correctly integrates with the REST server for each endpoint.

FR5: I would perform unit-level tests to verify any helper methods used by the path-finding algorithm function correctly on their own. I would then perform integration-level tests to see verify that the path-finding algorithm correctly uses these helper methods to generate a valid path (defined by the sub-criteria).

FR6: I would perform integration-level tests to verify that the pathfinding algorithm is only invoked for orders deemed valid by the order validation. Given a set of orders with different statuses, I would check that the paths file only contains paths for orders that I marked as valid.

NFR1 and **NFR2** can be verified with some form of statistical testing. **NFR4** and **NFR5** can both be verified by manual inspection of the code and tests, with a checklist for each.

4 Evaluation Of Test Approach

1. Comprehensive unit testing will help prevent algorithmic faults from being detected in the later stages of development during integration-level and system-level testing. An occurrence like this could result in development being paused to investigate what is causing it. If it were detected earlier it would be less of an inconvenience, and easier to find. Unit testing requires mock data, however, so ensuring that this is representative of the real world is important.

2. It is difficult to guarantee seamless integration between the rest server and the application. If the server goes offline (which is out of our control), the application cannot integrate with it, thus causing the integration tests to fail. In addition, it is difficult to test how our system integrates with the server when handling heavy loads as we cannot perform stress testing on it.

3. The approach of mocking responses will allow us to invoke certain exceptions that are hard to emulate in integration-level tests, thus allowing us to be more confident in the robustness of our REST client.

4. Code inspections are subjective, and different people may have different ideas on what qualifies as "readable" code or different people may have different opinions on what qualifies as an appropriate JavaDoc comment potentially leading to inconsistencies in code quality and disagreements, thus having a set criterion is important to uphold consistency. Team conflicts shouldn't be an issue however as this is a solo project.

5. Statistical Tests are effective in avoiding flukes or executions occurring that occur below 60 seconds by chance. The approach outlined will allow us to confidently say that our program meets the performance requirements outlined.