TESTING TECHNIQUES AND RESULTS

A variety of testing techniques was considered for each requirement, including systematic functional, structural, and model-based testing approaches. Below are details of the approaches used for testing each requirement and the reasoning behind them.

Note: Integration tests between modules and interfaces were accomplished with mainly a bottom-up integration approach, though there were slight aspects of a sandwich integration approach used as well. This was the most appropriate approach, as error identification is more efficient compared to other approaches. This allows for more thorough code development and testing and reduces the errors faced at high levels of development.

The big bang approach was not considered, as modules are not tested before integration, and could cause numerous faults to appear late in the development of the system. This would significantly increase the difficulty in debugging and lengthen the time required for testing. The top-down approach was not chosen as many stubs are required and the bottom-level modules may not be tested to meet the expected level of requirements. This may increase the time required for testing and thus lengthen the critical path in the project schedule. Finally, the sandwich approach was not fully utilized as the cost is too high and could cause various risks in terms of the schedule and quality of development. Hence, in conclusion, the bottom-up approach was taken instead.

Requirement 1: The system shall fetch all data required by the drone from a REST server. The system shall also verify all data fetched from the API.

Range of techniques: This requirement is an integration level requirement, as it requires interfacing with the REST API and classes from other modules to create Java Objects. Hence, integration level testing was used throughout the creation phase of DevOps. Systemic functional testing helped efficiently partition the input space and created test suites accordingly. For example, test suites for the URL validation were generated by firstly grouping valid and invalid URLs together. This included grouping valid URIs and IRIs together, whilst invalid inputs like empty strings and invalid IRIs were grouped. Various other approaches were considered as well:

Firstly, compared to random testing, systemic functional testing categorizes invalid and valid inputs according to the specification and reduces the number of test cases required. Secondly, combinatorial testing was not necessary to be used as there is only one input to create test suites. Thirdly, model-based testing was avoided as it has a steeper learning curve than the other approaches, and with an existing schedule risk, the cost of implementing it would be too high. Conversely, structural testing was used alongside systematic partition testing, as they are complementary to each other. Systematic partition testing helps ensure the correctness of the results whilst structural testing, specifically mutation testing, helps uncover hidden faults that are missed by systematic partition tests. For example, after systematic partition testing has been run, code coverage will be calculated to find any missing coverage. This way, inadequate test suites could be discovered, and more test cases could be added.

Scaffolding was needed for both systematic partition testing and structural testing to simulate the REST server and the data contained. Synthetic data was created to verify the interface between the system and the REST server during both tests. This allowed me to evaluate the correctness and robustness of the system when fetching data from the REST server for systematic partition testing. Structural testing was completed via IntelliJ IDEA's built-in "run with code coverage" functionality, which allowed me to view the branch and method coverage in each class.

Evaluation criteria: I will evaluate the systematic partition tests on a pass-or-fail basis, as its testing framework reflects the system's correctness for each input. Additionally, structural testing will be evaluated via method and branch coverage instead of path coverage and will be considered sufficient if there is above 70% coverage. Branch coverage provides more confidence in comparison as it checks if both paths for conditionals are taken. There are twice as many branches as conditionals, hence more code will be executed during the tests.

Results of the tests: *Refer to the "test results logging" document.*

Evaluation of the results: From the results of both tests above, I observed that most test cases in my test suites managed to catch most of the boundary or edge cases. Although one test case, whose URL input included white spaces, didn't pass, it was found to be caught in the second test suite for being unresponsive. This has increased my confidence in my system being robust enough for the current test case. The result from the structural tests further increases the confidence level. Even though the branch and method coverage over the entire system is very low, the coverage for the RestClient and CentralArea classes is high. Since this is an integration test that only involves those two classes, the branch and method coverage for those tests individually are high. At 100% branch and method coverage, this raises the confidence that my test suites have effectively caught most faults.

Requirement 2: The system shall validate all orders received before generating a flight path for the drone.

Range of techniques: This requirement is an integration level requirement, as it interacts with both the REST server and the system's API. Thus, this is an integration level test. Following the partition principle from chapter 3 of the Y&P textbook, I have divided each validation requirement for orders into unit level requirements and tests. This includes the following:

1. Each order must contain between 1 to 4 pizzas.
2. Each order must contain pizzas from the same restaurant.
3. Each order must contain pizzas from the menu of the given list of restaurants.
4. Each order must contain valid transactional details. The number, expiry date and CVV of the credit card must all be validated.
5. Each order must contain the correct stated total cost. This includes the total price of all pizzas ordered and the delivery fee of £1.

Although this helps ensure all inputs are guaranteed to be covered by tests, it has shown that a large combination of inputs must be tested. Hence, I decided on using category partition testing for each unit level requirement, pairwise combinatorial testing for validating a complete singular order, and functional testing for integration level testing. Additionally, structural testing will be used to further raise confidence in the test suites.

Category partition testing is appropriate for unit level tests as it helps eliminate contradictions with properties and reduces combinations for error and single classes. Pairwise combinatorial testing is also suitable for completely validating an order, as it reduces the time required to complete the execution of a test suite and costs less scaffolding. Systematic partition testing and model-based testing are both not suitable since there are too many possible inputs and environments to be covered. Both test approaches are too costly and would require more test cases to achieve the same amount of code coverage. Some scaffolding was used for this section, including data created to simulate participating restaurants and customer orders.

Functional testing is appropriate to test the interfacing between the orders module, REST server and system API. Since lower levelled tests have already been conducted at this point, black box testing using functional tests would suffice to validate the requirement. Structural testing would be suitable to evaluate the completeness of test suites executed and increase confidence levels. Model-based testing would be too costly in this scenario and could increase risks on the project schedule and quality of development.

Evaluation criteria: As this requirement concerns the robustness of the system and the safety of customers, the testing approach should be pessimistic, as suggested in chapter 2 of the Y&P textbook. This will induce a stricter evaluation criterion compared to the requirements. Hence, I will evaluate the category partition tests, pairwise combinatorial tests and integration level functional tests on a pass-or-fail basis. Structural tests will require a minimum of 85% code coverage, focusing mainly on method and branch coverage.

Results of the tests: *Refer to the "test results logging" document.*

Evaluation of the results: From the results above, I observed that all edge cases presented in the test suites have been successfully detected by the system. This increases confidence in the system's robustness. Additionally, the structural tests have revealed test cases which were missed in a few test suites. Although the code coverage appears to be low, this is due to various unused methods and classes, as it is an integration test and only parts of the system will be used. Additionally, some branch statements not covered included checking for null objects, which I was unable to test due to limited resources. Thus, after recalculating the total code coverage for the relevant classes, a method coverage of 93% and a branch coverage of 88.5% were achieved. Bearing all the mentioned considerations in mind, combinatorial testing does give me confidence in the code developed and structural testing adds to that by covering missed test cases.

Requirement 3: The mean time for the system to generate flight paths for all orders in a day and produce the resulting files in JSON format shall not exceed 60 seconds.

Range of techniques: This requirement is a system level requirement, as it is a non-functional requirement which is a measurable attribute of the code. Thus, system level testing will be used throughout the verification phase of DevOps to verify this requirement. This test focuses on the response speed of the system, which implies that the system must be fully functional before verification could be done. Thus, this requires a black-box testing approach. Assuming that the system's functionality has been fully tested, performance testing would be the most suitable approach for this requirement.

Other testing approaches like systematic functional, combinatorial, model-based and structural are not appropriate as they evaluate the system's functionality instead of ensuring it performs well enough to meet user expectations. Other non-functional tests like load testing to check the throughput or compatibility tests to check the interoperability of the system were not executed due to the lack of access to resources. However, if there were sufficient resources, I would include checking the process time and variety in system environments in addition to the throughput and interoperability when conducting the performance test.

For the current performance test, some scaffolding is required to simulate a real-world environment. This includes a list of participating restaurants, customer orders, no-fly zones, and a central area. A simulation of the hardware of the drone is also necessary to ensure any constraints in a real-world setting are imposed as well. This helps ensure the drone will perform as expected during its actual

launch. Some scaffolding will also be required to track the amount of time that passes between the start and end of system execution.

Evaluation criteria: Since this requirement has a medium priority, the testing approach taken would be a simplification of the system properties. The performance test will be evaluated on a pass-or-fail basis, depending on if the system completes execution within 60 seconds.

Results of the test: *Refer to the "test results logging" document.*

Evaluation of the results: From the results above, it is observed that at average user times, the system can perform in under 60 seconds. This raises the confidence that the system meets user expectations. Although the performance test is effective, this test only covers a predicted average number of orders. With more resources, I would try to cover a range of amount of users and validate the response speed of the system in those instances.

Additional note

The restriction principle was applied to validate requirement 2 as well. Since the system under test is still under development, there are multiple simplifications to the system. Below are a few clarifications to the testing detailed in the "test implementation" and "test results logging" documents:

- Only the following credit cards are accepted for payments: VISA 16 and MasterCard.
- Each order can only consist of pizzas ordered from the same singular restaurant.
- Each order can only consist of pizzas that are existing menu items from a restaurant.