Test Planning Document

Requirements, their priorities, and justifications

In total, 5 requirements were chosen for the Test Planning Document. This document was prepared using the example given in the course Software Testing. Other requirements can be found in the Requirements document. This specific combination was chosen for the Test Planning document due to its diversity – some are functional, some are non-functional, some are high priority, and some are lower, some require lots of work and scaffolding to test, some are more straightforward, some are system level, others are lower level requirements.

The requirements chosen for this document are:

- **R1**: *The orders shall be retrieved from the REST server must be validated (Credit card number, expiry date, cvv number, total cost of the order, pizza names, pizza count, pizza supplier). The precise validation details are explained in the Requirements document.*
- **R2**: *The system should aim to have a runtime of 60 seconds or less before terminating. This runtime should be achieved on a machine that has its system specifications similar to or better than the machine student.compute.inf.ed.ac.uk when it is lightly loaded (i.e., when the who command lists fewer than ten users using the machine).*
- **R3**: *The drone must enter pre-defined no-fly zones where people are crowded together to minimize the consequences of impact in case of hardware of software failure.*

Priority and pre-requisites

R1: *The orders shall be retrieved from the REST server must be validated for correctness (Credit card number, expiry date, cvv number, total cost of the order, pizza names, pizza count, pizza supplier). The precise validation details are explained in the Requirements document.*

Priority: High.

Priority justification: The orders must be up to date during the system's execution. Therefore, they shall be retrieved from the REST server during each execution (the other team is responsible for making sure the REST server contains the most up-to date orders). This way, only the most up-to date orders will be processed for delivery and any changes will be reflected in the system (for example cancelled orders will not be processed and delivered).

Importantly, if the system delivers orders that are not validated, it may cause financial difficulties to the owners of the PizzaDronz service. For example, users could input arbitrary credit card

details and get a pizza delivered without payment – so it is essential to validate all details before delivering pizza to an order.

The requirement is of high importance and can be divided into subproblems according to the partition principle:

- Firstly, the functionality to deserialize JSON format data into objects can be unit tested without accessing the server to ensure that the deserialization does not rely on server connection. This requires additional setup to imitate server connection. The JSON files for unit testing can be generated manually.
- A later suit of tests that ensure the deserialization integrates well with the data access from the REST server. There should also be inspection so that only data from the server is used instead of local files.
- Then the functionality to validate orders can be tested using unit tests. The orders shall be hardcoded into Order objects (as if they are already deserialized). This makes sure the order validation works without reliance on the REST server or deserialization. This part can be further partitioned:
    - Unit tests to make sure the credit card validator works as expected.
    - Unit tests to make sure the rest of order validation works as expected.
    - Integration tests to make sure the card validator works as expected together with the rest of order validation.
- Finally, a suite of tests to make sure the order deserialization from the REST server functionality integrates well with the order validation functionality. This will make sure the requirement R2 is satisfied. There should be inspection that only orders from the REST server are used.

R2: *The system should aim to have a runtime of 60 seconds or less before terminating. This runtime should be achieved on a machine that has its system specifications similar to or better than the machine student.compute.inf.ed.ac.uk when it is lightly loaded (i.e., when the who command lists fewer than ten users using the machine).*

Priority: Low

Priority Justification: While it is more convenient for users to have fast runtime, it is possible to have workarounds for now (for example computing all the orders for the day at midnight, etc.) to save the budget and time resources for higher priority requirements.

This is a measurable attribute of the code, so we need means to measure this:

- This can only be tested for the completed system, so this is a system level test that occurs late in the process.

- To verify, we will need to generate synthetic data containing orders for a given day and the means to run tests on this.
- To validate and verify, we will need logging performance of the system.
- This suggests the following tasks need to be scheduled into the testing:
    - Generating synthetic data to test the runtime of the system.
    - Designing the logging system to capture the performance of the whole system.
    - Ensuring the captured performance matches the requirements outlined in R2.
    - Designing and implementing the analytics intended for the order data.
    - Feeding collected data back into the early testing.

R3: *The drone must enter pre-defined no-fly zones where people are crowded together to minimize the consequences of impact in case of hardware of software failure.*

Priority: High.

Priority Justification: In some cases, the no-fly zones can include military/private areas or places where people are crowded. Drone entering such areas can cause legal issues (military areas), also incur financial costs due to fines of entering such areas and could provide important safety concerns if the drone malfunctions or crashes inside these zones, damaging property or hurting people.

This requirement is of high importance. The partition principle suggests this problem into subproblems. The subproblem is checking if two different line-segments intersect, one line segment being the drone's chosen move and the other line-segment is any of the no-fly zone edges (it makes sense because the drone enters a no-fly zone if and only if it crosses any edge of any no-fly zone). This subproblem can be tested primarily using unit tests. Finally, it is important to check that the drone picks a direction for a move out of 16 different available directions according to the requirements (i.e. does not fly into a no-fly zone), ensuring the requirement R1. Therefore, the direction choosing logic can be tested using a wide range of situations, focusing on the ones that are directly next to the no-fly zones. So overall, we can see two different tasks we need to schedule for the plan:

- Thorough unit tests for the line-segment intersection algorithm and inspection that there is no extraneous code that cannot be seen as implementing the requirement.
- A later suit of tests to check that the drone never chooses a direction that crosses a no-fly zone boundary using the line-segment intersection algorithm and checks that the result conforms to the specification. This suit of tests can be generated randomly, but that requires additional setup and tools.

Scaffolding and instrumentation

This section describes scaffolding and implementation that are needed in order to carry out the given tasks (and this may give result in more tasks to build scaffolding and instrumentation).

For our requirements:

- R1: This would require artificial JSON order data as well as a working REST server that contains (simulated) orders for testing.
    - The deserialization must not rely on the server connection, as mentioned in the earlier part. This requires additional scaffolding and instrumentation to imitate server connection. The JSON files for unit testing can be generated manually due to limited resources allocated for this project.
    - For the server access integration with deserialization tests, we need a working and accessible REST server, that has the correct endpoint and contains orders in a JSON format. This is also required for the integration of REST server order deserialization with validation tests.
    - For credit card validation methods, we need generate mock (valid and invalid) credit card numbers, expiration dates (with correct and incorrect formats, expired and still valid dates) and (valid and invalid) cvv numbers.
    - For testing the rest of order functionality, we can hardcode a suite of valid and invalid pizza orders and their prices.
    - For the credit card integration with the rest of order validation tests, we need to hardcode a suite of mock Order objects into the tests. This makes sure the order validation functionality works with already deserialized orders without access to the REST server. It is possible to reuse the previously generated data for the Order objects.
    - A process or tool for is necessary verifying the accuracy and completeness of the test data.

- R2: This would require synthetic data for the whole system.
    - The data can be reused from other system tests.
    - To measure the performance fairly, access to a computer with the required system specification is required. Currently it is possible to use the university machines to test this requirement, as they have the satisfy the system specification.
    - Varying sizes of data for testing the whole system. These can be generated automatically and placed on the REST server.
    - A process or tool for is necessary verifying the accuracy and completeness of the test data.
    - We would also need to schedule the instrumentation to log the activity in the system and tools for analysis.

- R3: This would require data for the tests.
    - There must be a wide range of coordinates to test for the drone to make sure the requirement is satisfied. This would require some effort generating both the coordinates of the drone and no-fly zones and checking the output meets the specification. The coordinates can be generated manually, to ensure numerous edge cases are covered, as well as some usual cases that are expected to commonly occur. Additionally, some randomized coordinate generator can be used to ensure the requirement is satisfied even with some potentially unexpected inputs, but this requires additional scaffolding that needs to be scheduled early in the development process.
    - The inspection of the line-intersection algorithm does not need any scaffolding or instrumentation. However, the algorithm must be thoroughly tested using unit tests, using a wide range of scenarios, including edge cases. This must also be planned even earlier, since a large part of the system depends on this algorithm.
    - A test environment that includes data on the locations and characteristics of the no-fly zones is also necessary.

Process and Risk

R1. This task could take around a day to complete. It would require access to the REST server prepopulated with at least a few sample orders, as well as a requirements document with precise validation details. This requirement could be placed in the elicitation and analysis stage of the SRET lifecycle process because it involves collecting and clarifying the needs of various stakeholders. It is also possible to include it in the design and implementation stage, as it specifies the validation process the software should follow.

The risks for this requirement involve problems with the REST server: for example, what if the information there is incomplete or incorrect? This could potentially lead to invalid orders being accepted or rejected. So it is important to make sure to implement additional checks or verification steps to make sure the data on the REST server is complete and accurate. It is also possible that the REST server fails or becomes unavailable, so it is important to have alternative sources or contingency plans. On the other hand, it is also possible that the validation process could be overly complex or take too much time, leading to delays and inefficiencies in the project.

R2. This task could take around a week to complete, depending on the complexity of the software and the availability of resources. It also requires access to a computer with the necessary system requirements, as well as the complete software to be tested. This requirement can be placed in the testing stage of the SRET lifecycle process because it specifies a performance target the

software is supposed meet. It could also be included in the design and implementation stage, as it may influence the design and optimization of the software.

Like in R1, it is possible that the REST server might fail or become unavailable, so it is important to make sure to have alternative sources or contingency plans. It is also possible that the data for the performance tests is unrepresentative of the real conditions, leading to inaccurate results. So it is important to consider a wide range of scenarios and configurations to make sure the system performs well under different conditions. To mitigate the risk of an insufficiently representative testing environment, it is possible to test the software on different machines or in different usage scenarios to ensure that it performs adequately in a wide variety of conditions.

R3. This requirement could possibly take around several days to complete, depending on the software complexity. It would require access to data on restaurant locations and the no-fly zone borders, as well as the details about those no-fly zones (for example, whether they can exist outside of the central area, etc.). This task can be placed in the elicitation and analysis stage of the SRET lifecycle process because it involves identifying and addressing safety concerns. It could also be included in the design and implementation stage, as it specifies an important constraint the software must follow.

The main risk here is that the navigation design process might be overly complex and take too much time, leading to delays and inefficiencies in the project. Additionally, the software might malfunction or fail, which would lead to unintended or unsafe drone flight in banned areas. So, it is possible to consider streamlining the process or identifying opportunities for automation or optimization, but ensuring the software is tested extensively to identify any possible issues.