*Outline of the Software Being Tested*

The software tested in this project is PizzaDronz application as part of the Informatics Large Project course. The system retrieves pizza orders from the REST server, validates them, makes a drone navigation plan around no-fly zones and outputs the required results in the 3 output files. The precise details can be found in the requirements document or in the coursework specification. The repo of the software can be found ***here***.

*1. Analyze requirements to determine appropriate testing strategies [default 20%]*

1.1. Range of requirements, functional requirements, measurable quality attributes, qualitative requirements, …

A wide range of requirements was identified in the Requirements Document. The following combination of attributes in the Requirements Document was identified to be the most diverse and important:

* **Safety Requirement 1.1.4**: The drone should return close to Appleton Tower (within 0.00015 degrees distance) before running out of battery energy.
* **Correctness Requirement 1.2.4**: The orders 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.
* **Liveness Requirement 1.3.2:** In case the system is started in an unsuitable state (i.e., invalid input described in requirement 1.2.2), the system shall terminate with an appropriate error message without creating the output files.
* **Efficiency Requirement 2.1:** The system should aim to have a runtime of 60 seconds or less before terminating when processing no more than 7050 orders. 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).

1.2. Level of requirements, system, integration, unit.

The credit card validation for the Correctness Requirement 1.2.4 can be tested at **unit level** as the credit card validation functionality is likely to be grouped in one class and can be tested for correctness as an isolated unit of code. **Integration level** tests include testing if the credit card validator integrates well with the rest of the validators, and if the combined validation functionality integrates well with data access to the REST server (where the deserialization can be unit tested in isolation as well, before applying integration with the REST server tests). **System level** tests would be required to test the Efficiency Requirement 2.1, as the whole system would need to be run to measure the overall performance.

1.3. Identifying test approach for chosen attributes

TO DO

1.4. Assess the appropriateness of your chosen testing approach

The performance tests are difficult to perform due to the lack of order data are when generating realistic loads for testing. In addition, the Rest server access time may vary and thus influence the performance tests. Furthermore, the Rest server data is not currently modifiable, as it has been created by a different team working on this project. This makes it more difficult to test the correctness requirement 1.2.4. using integration tests with the Rest server for data retrieval.

*2. Design and implement comprehensive test plans with instrumented code [default 20%]*

2.1. Construction of the test plan

The attached Test Planning document was constructed considering the characteristics and requirements of the project, together with the scaffolding and the risks and challenges that need to be addressed. A diverse sample of requirements was analysed and the testing for each requirement was placed into the SRET process lifecycle and could be similarly extended to a larger collection of requirements in the future.

2.2. Evaluation of the quality of the test plan

The Test Planning document provides an optimistic plan to ensure adequate testing but acknowledges the possible risks. The main risks include issues with the REST server (it might fail or could contain incomplete data), the risk of test samples being unrepresentative of real conditions, leading to inaccurate performance and functional results, and that some of the processes might become overly complex and take too much time, leading to delays in the other parts of the project. The Test Planning document also contains suggestions how to mitigate these issues, while ensuring the testing is adequate.

2.3. Instrumentation of the code

The scaffolding includes test environments, test data, and tools or processes for implementing and evaluating the requirements. The instrumentation provides access to the necessary resources and information to thoroughly test the requirements and verify the accuracy and reliability of the results. It is comprehensive and covers a range of different conditions and scenarios, which helps to ensure that the requirements are tested in a thorough and reliable manner.

2.4. Evaluation of the instrumentation

The Test Plan Document currently contains extensive plans for scaffolding to generate data. While the randomly generated data together with manual testing can provide more thorough testing, it might require more time and effort than it has been allocated for this project, which might decrease the quality of the project. The possible improvement could be generating data on the server – currently the data on the server is not modifiable due to the restrictions by the Informatics Large Project course organisers. The data could be simulated by creating another server for testing purposes, but this requires additional finances to host the test server and populate it with synthetic data.

*3. Apply a wide variety of testing techniques and compute test coverage and yield according to a variety of criteria [default 20%]*

3.1. Range of techniques

A wide range of tests was performed: **unit tests** (*TestCreditCardValidation*, *TestLngLat* and others), **integration tests** (*TestOrderIntegrationWithRest* and others) and **system tests** (*TestSystem*). Systematic functional tests can be seen throughout the unit and integration test classes. Additionally, there have been elements of **structural testing** implemented (for example, **equivalence partitioning** the orders by valid/invalid fields and testing representative values from each class to ensure validation is working correctly), as well as elements of **boundary value analysis** (for example, by testing the ranges of accepted credit card numbers or central area methods). Need to write about Model! The performance was tested by recording the time just before the system starts and just after the system finishes computation and calculating the difference between these – while this diverges from the planned testing, it is simpler and still allows testing of the performance requirement.

3.2. Evaluation criteria for the adequacy of the testing

The tests performed are towards the optimistic side. For example, in *TestDeserialize* unit tests, which deserializes the given samples of mock orders that are stored locally, only smaller samples (that include ≈5 orders) are tested thoroughly for correctness, whereas the larger samples are tested for the size and selected orders. This means that if there are errors that only occur if the file is large enough and cause incorrectly deserialised orders, they might be missed in these tests.

3.3. Results of testing

It was discovered that the system robustness was low when unit testing credit card validation methods with null values, which meant if the REST server data was incomplete/distorted, it would crash the whole system. These issues were fixed as they were identified by the tests by implementing guards against null values (so that they are treated as invalid). Additionally, there were issues with the output files, as the starting location was not written into the output files and the tests to start at the Appleton Tower did not pass. This was fixed by saving the starting location at the very start of the pathfinding process.

3.4. Evaluation of the results

*4. Evaluate the limitations of a given testing process, using statistical methods where appropriate, and summarise outcomes. [default 20%]*

4.1. Identifying gaps and omissions in the testing process

4.2. Identifying target coverage/performance levels for the different testing procedures

4.3. Discussing how the testing carried out compares with the target levels

4.4. Discussion of what would be necessary to achieve the target levels.

*5. Conduct reviews, inspections, and design and implement automated testing processes. [default 20%]*

5.1. Identify and apply review criteria to selected parts of the code and identify issues in the code. [default 20%]

5.2. Construct an appropriate CI pipeline for the software

First, I would set up some version control system (like Git) to track all changes to my project and some build server (like Jenkins) to automatically build the code and run tests whenever a change is made to my project. The build server will pull the latest code from the version control system each time a change is pushed and run series of tests on the code, whenever it is built. These tests would include different levels: unit, integration, system. And then, if all tests pass, the build server will deploy the code to a staging or production environment, while also notifying the developer(s) (possibly by email). Alternatively, if not all tests pass, the build will fail, and the developer(s) would be notified accordingly.

5.3. Automate some aspects of the testing

For safety-critical requirements, more extensive testing in the CI pipeline is required to make sure the requirements are fully satisfied. While it is generally desirable to include as many tests as possible, the available resources are limited, so testing for the safety-critical requirements need to be prioritised over other others in the CI pipeline. TO FINISH

5.4. Demonstrate the CI pipeline functions as expected.

The CI pipeline is expected to identify issues with the code and alert developers. One of the issues could be failed tests or syntax errors that would make the build server fail the build. In both cases the developer would receive notification, investigate the issue, fix it accordingly and re-run the build. Additionally, there could be misconfigured/incompatible dependencies or performance issues that can be identified by the CI pipeline during the build or deployment process.