Outline of the Software Being Tested

The software tested is a modified PizzaDronz application based on 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 3 files. For more precise details and additional files, the repository with all files mentioned in the portfolio is: https://github.com/ikleveckas/PizzaDronz.

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*. These include safety (req. 1.1.1 – 1.1.6), correctness (req. 1.2.1 – 1.2.11), liveness (req. 1.3.1 – 1.3.2), performance (req. 2.1), resource utilization (req. 2.2), reliability (req. 2.3, 2.4), data integrity (req. 2.5), interoperability (req. 2.6-2.7), maintainability (req. 2.8), among others described in the document.

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 Performance 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.

Performance tests could be run after the system is complete, by measuring the time it takes for the whole program with a set of mock orders. Similarly, liveness requirement can also be tested using system tests by providing invalid console input and checking result. Correctness requirement can be tested using unit/integration tests by creating a set of mock orders with predefined valid/invalid fields and expecting the correct results. The safety requirement can be tested while unit testing the navigation unit by ensuring the last coordinate of the drone is close (according to the definition) to Appleton Tower.

1.4. Assess the appropriateness of your chosen testing approach.

The performance tests are difficult to perform due to the lack of data when generating mock orders for testing. In addition, the Rest server access time may vary (for example due to internet connection on the device for testing) and thus influence the performance tests. Furthermore, the Rest server data is not currently modifiable, as it has been populated by the ILP Course Organisers for the whole student cohort.

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

2.1. Construction of the test plan

The *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, such as unreliable REST server, test samples being unrepresentative of real conditions that result in inaccurate testing 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, among others. 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

While the randomly generated data together with manual testing can provide more thorough testing, it might require too much time and decrease the quality of the project. Also, the data on the server is not modifiable as it has been set by ILP course organisers. The data could be simulated by creating another server, 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 were performed including unit tests, integration tests, system tests, systematic functional tests, structural testing (using equivalence partitioning and boundary value analysis), stress and performance testing. A set of mock orders was generated, and scaffolding was used to imitate server connection. Regression testing was used throughout the development of the system. See *Testing Evaluation Document* for a more detailed discussion of the testing techniques and approaches and see the *Test Folder* for the tests themselves.

3.2. Evaluation criteria for the adequacy of the testing

The tests performed are towards the optimistic side (as sometimes the errors could appear in large enough files). The evaluation criteria include statement, function, class, branch coverage and defect detection rate. For a more detailed discussion, see the *Testing Evaluation Document*.

3.3. Results of testing

Robustness issues were discovered 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. This was fixed by implementing guards against null values. Also, the starting location was not written into the output files. This was fixed by saving the starting location at the very start of the pathfinding process.

3.4. Evaluation of the results

JaCoCo library and mutation testing were used to measure the chosen evaluation criteria, as explained in the *Testing Evaluation Document*. 91% of statements, 97% of methods, 96% of classes, 82% of branches were covered and DDR was found to be 95%. This combination of high results helps improve the overall confidence in testing, however, it does not guarantee the system is free of defects.

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

Deficiencies: REST server populated with unmodifiable testing data, absence of realistic order data, lack of test cases for navigation algorithm (to ensure it follows optimal path, etc.), lack of testing on different platforms. Remedies: creating a custom server for testing, using anonymized order samples from restaurants or open sources, creating scenarios with known optimal paths, testing on DICE machines.

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

Target levels of 90% of statement, function, class, branch coverage, as it is easy to achieve and maintain, ensures most of the code is executed by the tests, providing good level of confidence the code is tested thoroughly. Achieving 100% coverage is not feasible with the given resources. Similarly, target of 90% of DDR as it means most mutants were detected and the test suite is unlikely to have undetected defects.

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

The *Testing Evaluation Document* discusses the results more rigorously. The target was achieved for statement, function, class coverages and DDR, as they all were higher than 90%. However, the branch coverage was 82%, lower than the target of 90%. This is because it difficult to test the fallback functionality (for example when REST server data is distorted), using the resources and time available.

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

The branch target levels can be increased by creating mock order, no-fly-zone, and restaurant files locally, introducing errors, using scaffolding in tests to simulate mock files as data retrieved from REST server. This should increase the branch coverage, as different outcomes for fallback checking would be triggered.

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%]

The *Navigation* class code follows the conventions of the project, such as using camelCase for variables and methods except for one variable name, which was fixed. The code has Javadoc comments for all public methods, however, some of the exceptions were not documented in the comments. Exceptions are generally handled correctly, however *navigateTo* method that is used in an outside class (*OrderProcessor*) raises a custom exception and must be handled with care. Code formatting and types are consistent.

5.2. Construct an appropriate CI pipeline for the software

The detailed steps can be found in the *Automated Testing Processes* document. These include setting up VCS repository, configuring Jenkins job, running Maven commands, and monitoring the deployed code.

5.3. Automate some aspects of the testing

It is desirable to include as many tests as possible, but the safety-critical requirements should be prioritised in the CI pipeline. A combination of unit, integration, system, and other tests (using tools described in *Automated Testing Processes* document) can be used to ensure rigorous testing.

5.4. Demonstrate the CI pipeline functions as expected

A more rigorous demonstration has been provided in the *Automated Testing Processes* document. Some issues discovered by the proposed CI pipeline include syntax, performance, functional errors.