

# Testing Evaluation

## Range of Techniques

A total of 104 tests were written for the PizzaDronz system. They cover a wide range of levels/techniques can be found in the test folder of the ILP coursework repository. The different levels of tests include:

- **Unit tests** (*TestCreditCardValidation*, *TestLngLat* and others) ensure the individual units of the system work correctly. For example, to test that the drone can move to the 16 specified directions in the distance specified in the Requirements Document, unit tests were used for this purpose.
- **Integration tests** (*TestOrderIntegrationWithRest* and others) were used to ensure the different subsystems can integrate well with each other. For example, integration tests were useful to check that 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 tests** (*TestSystem*). System tests were used to ensure that the whole system behaves correctly: validates the console input, retrieves data from the REST server, validates the orders and delivers them according to the restrictions specified in the Requirements Document. The final output files were all checked for correctness. It was also ensured that the system behaves according to the specification when invalid input values are presented. Additionally, performance tests have been added to test the full system runs under the 60s limit specified in the Requirements Document.

**Systematic functional tests** can be seen throughout the unit and integration test classes. The tests were an application of **systematic functional testing**, as they were focused on meeting the requirements from the requirements specified in the Requirements Document. Some of the tests were created using a method called specification-based testing, which help us confirm that the individual components of the project meet their functional requirements and ensures that the integration of different subsystems (like REST server retrieval into the deserialization mechanism) behaves as expected.

Another technique that was used in the testing process is **structural testing**. One example where this was used was by equivalence partitioning the orders by valid/invalid fields and testing representative values from each class to ensure validation is working correctly. **Boundary value analysis** was also used as one of the techniques in the testing process. For example, this was applied when testing the ranges of accepted credit card numbers and ensuring only the card numbers that lie strictly inside the ranges of MasterCard or Visa cards are accepted. This involved checking the

minimum and maximum values in the union of Mastercard/Visa ranges, the values just below the minimum value of the range and just above the maximum value of the range, and some values that are in the middle of the range, etc. Another example can be seen by checking central area methods work as expected (so that the point just inside the central area is treated as inside the area, etc.). The **performance** was tested by recording the system computation time. For this, a set of **mock orders** was generated with various sizes.

Additionally, in order to ensure the deserialization does not rely on the REST server connection, **scaffolding** was required to imitate the server connection. This was made by creating a set of mock orders in JSON format and saving the files locally in the testing depository. Then the URL address of the server was changed into a local address of my testing depository on the same machine as the tests are performed. This way, the deserialization was checked locally, without reliance on any internet connection. See TestDeserialize file inside the test folder for the tests themselves.

## Evaluation criteria for the adequacy of the testing

The tests performed are towards the optimistic side. For example, in TestDeserialize unit tests, only smaller samples are tested thoroughly, whereas the larger samples are tested for the size and selected orders. This means that if there are deserialization errors that only occur further in the process if the file is large enough and cause incorrectly deserialised orders, they might be missed in these tests.

- **Statement Coverage.**

One of the evaluation criteria used to determine the adequacy of the testing is statement coverage. It refers to the percentage of statements in the code that have been executed by the tests. It is a more accurate measure of how much of the code has been covered (compared to line coverage), as it considers the complexity of the code. To execute as many statements as possible,

- **Function coverage.**

Another criteria used to determine the adequacy of the testing is function coverage. This measures the percentage of functions (or methods in our case) in the system that have been executed by the tests. A high function coverage indicates that most of the functions have been tested.

- **Class coverage.**

Class coverage was used to measure of how much of the classes in the system have been executed by the tests. Like the other coverage metrics, it is used to evaluate the adequacy of the testing and identify areas of the system that need more testing. A class coverage of 100% means that all the classes in the system have been executed by the tests. However, as with line and statement coverage, a high class coverage does not guarantee that the system is free of defects or that it meets the requirements.

- **Branch coverage.**

Branch coverage was used to measure how much of the branches in the system have been executed by the tests. Branches are defined to be points in the code where a decision is made, so for example if-else or switch statements. Branch coverage is a valuable metric as it describes which branches of the code have been executed by the tests and which have not. It helps us to prevent unexpected results or system failures that may occur due to a failure in one branch of the code.

- **Defect Detection Rate.**

The idea was to create mutants (deliberate errors) in the code and count the percentage of discovered mutants by the test suite. A high Defect Detection Rate after mutation testing indicates that the test suite is reliable and capable of quickly identifying system flaws. It implies that the testing are sufficient and that it is less likely that the system has undiscovered flaws.

## **Evaluation of Testing Results**

The chosen evaluation techniques were applied to the project using JaCoCo library in IntelliJ IDEA. This library was configured to show the results for the chosen evaluation techniques in IntelliJ IDEA project. The Defect Detection Rate was evaluated by using mutation testing approach. The results are as follows:

- **Statement Coverage (achieved 91%).**

91% of the statements in the whole system were executed. All classes have between 80-100% statement coverage (with exception of Output – see Class coverage for explanation). This is a high result and provides some confidence that the tests are exercising a significant portion of the code and that the code has been thoroughly tested. However, it is important to mention that it does not guarantee that the system is free of defects or that it meets the requirements. Even if all statements are executed during the tests, there may still be defects in the system that the tests have not been able to detect. Therefore, the combination of other criteria was used to achieve higher confidence in the testing results.

- **Function Coverage (achieved 97%).**

97% of the functions were covered by the test suite. All classes have between 95-100% statement coverage (with exception of Output with 80% – see Class coverage for explanation). This is also a high result and helps ensure that most of the functions or subroutines have been executed at least once during the testing. The most significant omissions identified were with the methods used when the REST server fails. It was not possible to modify the REST server as it was set up by the ILP course organisers, which means it was not possible to run tests to check the resilience of the system.

- **Class Coverage (achieved 96%).**

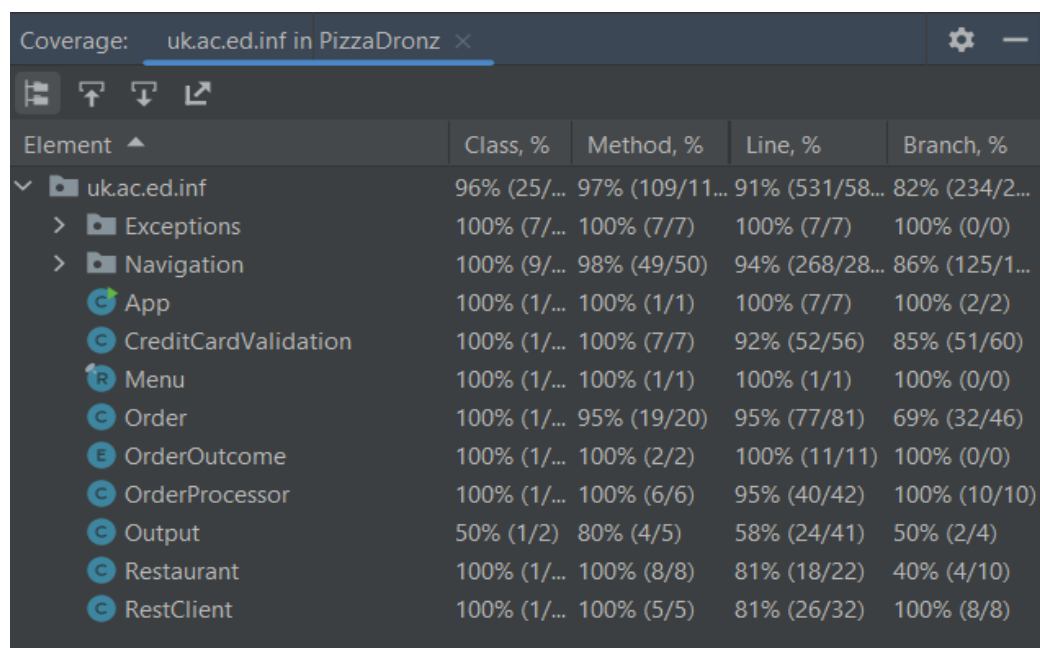
Class coverage of 96% was achieved by the test suite. All classes have 100% coverage, except the Output class. The problem with the Output class is that it contains a private ListSerializer class, which was only used in the outside Jackson library to achieve a custom serialization of deliveries into the output file. However, all other classes were tested by the test suite, which improves the confidence in the testing results. On the other hand, this criterion alone does not help guarantee that the system is free of defects or that it meets the requirements.

- **Branch Coverage (achieved 82%).**

Branch coverage of 82% was achieved by the test suite. This is a lower percentage than in other criteria, with Order and Restaurant classes having the lowest branch coverage of 69% and 40% respectively. This was because the branches for fallback functionality were not executed, as it was not possible to invalidate the REST server data due to the aforementioned reasons. However, overall, this level of branch coverage helps increase confidence in the testing by ensuring that all possible outcomes of a decision have been tested, thus reducing the risk of unexpected results or system failures.

- **Defect Detection Rate (achieved 95%).**

It was discovered that the defect detection rate was 95%. This was measured by creating mutants (small deliberate errors) in the code and counting what percentage of the mutants was discovered by the test suite. This indicates that the test suite is effective in detecting defects in the system that were introduced deliberately, and so it is likely to detect real defects as well.



Element	Class, %	Method, %	Line, %	Branch, %
uk.ac.ed.inf	96% (25/...	97% (109/11...	91% (531/58...	82% (234/2...
Exceptions	100% (7/...	100% (7/7)	100% (7/7)	100% (0/0)
Navigation	100% (9/...	98% (49/50)	94% (268/28...	86% (125/1...
App	100% (1/...	100% (1/1)	100% (7/7)	100% (2/2)
CreditCardValidation	100% (1/...	100% (7/7)	92% (52/56)	85% (51/60)
Menu	100% (1/...	100% (1/1)	100% (1/1)	100% (0/0)
Order	100% (1/...	95% (19/20)	95% (77/81)	69% (32/46)
OrderOutcome	100% (1/...	100% (2/2)	100% (11/11)	100% (0/0)
OrderProcessor	100% (1/...	100% (6/6)	95% (40/42)	100% (10/10)
Output	50% (1/2)	80% (4/5)	58% (24/41)	50% (2/4)
Restaurant	100% (1/...	100% (8/8)	81% (18/22)	40% (4/10)
RestClient	100% (1/...	100% (5/5)	81% (26/32)	100% (8/8)

Figure 1. Coverage results in IntelliJ